НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Автоматики та управління в технічних системах

«На правах рукопису»
УДК 658.5_____

«До захисту допущено»
Завідувач кафедри
_____ О. І. Ролік
«___»_____20__ р.

Магістерська дисертація

на здобуття ступеня магістр

зі спеціальності 151 автоматизація та комп'ютерно-інтегровані технології

на тему: «Підсистема підтримки життєвого циклу програмного забезпечення в межах програмного сервісу»

Виконав:
студент ІІ курсу, групи ІА-61м
Майер Ілля Сергійович _____

Керівник:
декан ФІОТ, д.т.н., професор
Теленик С. Ф. _____

Рецензент:
професор ФТІ КБІ, д.т.н., професор
Качинський А. Б. _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2018 року

РЕФЕРАТ

Дисертація освітньо-кваліфікаційного рівня “магістр” на тему «Підсистема підтримки життєвого циклу програмного забезпечення в межах програмного сервісу»: 140 с., 40 рис., 23 табл., 3 додатки, 37 джерел.

Об'єкт дослідження - розробка підсистеми підтримки життєвого циклу програмного забезпечення в межах програмного сервісу.

Мета роботи – розробка підсистеми підтримки життєвого циклу програмного забезпечення в межах програмного сервісу. Проаналізувати існуючі методології розробки ПЗ, запропонувати новий підхід до розробки ПЗ, спроектувати систему, що надає можливість підтримки життєвого циклу обраним підходом та підсистеми для взаємодій замовників та розробників.

Наукова новизна дослідження полягає в тому, що реалізовано проект системи, що надає можливість підтримки життєвого циклу запропонованого підходу та реалізації підсистеми для взаємодій замовників та розробників.

При розробці компонентів системи використовувалися сучасні технології як ASP.NET Core та EntityFramework Core.

Прогнозні припущення про розвиток дослідження – застосування метода підтримки життєвого циклу програмних засобів у сферах які містить в своєму складі програмні засоби.

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, РОЗРОБКА, СЕРВІС, ПРОВАЙДЕР, ЗАМОВНИК, ЖИТТЄВИЙ ЦИКЛ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

ABSTRACT

Dissertation of the educational qualification level "Master" on the topic "Subsystem of life cycle support of software within the software service": 140 p., 40 figures, 23 tables, 3 annexes, 37 sources.

The object of the research is the development of a subsystem supporting the software life cycle within the software service.

The purpose of the work is to develop a subsystem supporting the software life cycle within the software service. Analyze existing software development methodologies, offer a new approach to software development, design a system that allows you to support the lifecycle of the chosen approach and subsystems for customer and developer interactions.

The scientific novelty of the research is that the project of the system is implemented, which enables to support the life cycle of the proposed approach and implementation of the subsystem for interactions between customers and developers.

The development of system components used modern technologies such as ASP.NET Core and EntityFramework Core.

Foreseeable assumptions about the development of the study are the use of a method to support the software lifecycle in areas that contain software components.

SOFTWARE, DEVELOPMENT, SERVICE, PROVIDER, CUSTOMER, SOFTWARE LIFE CYCLE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення.

ПП — програмний продукт.

ТЗ — технічне завдання.

СПЗ – сервіс програмного забезпечення.

CS – референсна реалізація системи (CleanSlate).

XP – екстремальне програмування (eXtreme Programming).

RAD – швидка розробка додатків (Rapid Application Development).

RUP – раціональний уніфікований процес (Rational Unified Process).

UI – графічний інтерфейс (User Interface).

RPC – віддалений виклик процедур (Remote Procedure Call).

AMQP – протокол черги повідомлень (Advanced Message Queueing Protocol).

АРМ – автоматизоване робоче місце.

IoC – інверсія контролю (Inversion of Control).

EF – засіб для відображення реляційних даних у об’єкти (Entity Framework).

НФ – нормальна форма.

TDD – розробка через тестування (Test-Driven Development).

ВСТУП

Процес сучасної розробки програмного забезпечення орієнтований на життєвий цикл програмного продукту. Всі існуючі в даний час технології, методики та стандарти безпосередньо або опосередковано стосуються або регламентують етапи життєвого циклу, як за функціональним наповненням, так і за змістом.

Процес розробки програмних систем тісно пов'язаний з областю управління проектами, тому що будь-який програмний продукт є унікальним результатом. Від організації цього процесу безпосередньо залежать основні характеристики реалізації програмного проекту – терміни виконання, запланований бюджет, якість готового продукту.

Але професійне управління проектами саме по собі не може забезпечити досягнення зазначених характеристик. Важливу роль має архітектура системи, досвід учасників команди розробки, а також документування протягом всього життєвого циклу програмного забезпечення.

Програмне забезпечення розроблюється все більшими темпами з кожним роком, оскільки рівень інтеграції технологій у наше життя теж зростає. Розвиток апаратного устаткування призвів не тільки до збільшення потужностей та зменшення розмірів, але і до уніфікації виконавчих пристроїв. Це дозволило розроблювати програмні модулі що можна використати повторно і зменшити витрати.

Все це призвело до збільшення попиту саме на програмні розробки і саме цей попит призвів до виявлення проблем в процесах його розробки, оскільки все більше молодих і не досвідчених спеціалістів було залучено, а процес розробки не дозволяв кваліфікованим кадрам підтримувати рівень якості на високому рівні.

Саме ці перелічені чинники призвели до ідеї спрощення та уніфікації процесу розробки програмних засобів та можливість якісної підтримки протягом їх життєвого циклу.

Основні принципи програмного сервісу розглядалися на IV Міжнародній науково–практичній конференції з інформаційних систем та технологій — Summer InfoCom 2017, м. Київ, 1-2 червня 2017 р. (додаток А). За результатами роботи було опубліковано наукову статтю в міжнародному науковому журналі «ІНТЕРНАУКА» № 8 (48) / 2018 1 том (додаток Б).

Метою роботи є розробка підсистеми підтримки життєвого циклу програмного забезпечення в межах програмного сервісу. Проаналізувати існуючі методології розробки ПЗ, запропонувати новий підхід до розробки ПЗ, спроектувати систему, що надає можливість підтримки життєвого циклу обраним підходом та підсистеми для взаємодій замовників та розробників.

Відповідно до мети роботи необхідно вирішити такі завдання:

— розглянути поняття про життєвий цикл програмного забезпечення;

— провести проектування системи для взаємодії замовників та розробників;

— описати реалізацію системи;

— провести технологічні розрахунки системи;

— розробити стартап-проект.

Об’єкт дослідження — розробка підсистеми підтримки життєвого циклу програмного забезпечення в межах програмного сервісу.

Предмет дослідження — проект системи, що надає можливість підтримки життєвого циклу запропонованого підходу та реалізації підсистеми для взаємодій замовників та розробників.

Наукова новизна дослідження полягає в тому, що реалізовано проект системи, що надає можливість підтримки життєвого циклу запропонованого підходу та реалізації підсистеми для взаємодій замовників та розробників.

Магістерська дисертація складається зі вступу, п’яти розділів основної частини, висновків, списку використаних джерел та додатків.

1 ПОНЯТТЯ ЖИТТЄВОГО ЦИКЛУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Загальні поняття про життєвий цикл програмного забезпечення

Поняття «життєвий цикл» передбачає щось як таке, що народжується, розвивається і вмирає. Подібно живому організму програмні вироби створюються, використовуються і розвиваються в часі.

Життєвий цикл програмного забезпечення включає в себе всі етапи його розвитку: від виникнення потреби в ньому до повного припинення використання внаслідок морального старіння або втрати необхідності вирішення відповідних завдань [32].

Можна виділити кілька фаз існування програмного продукту протягом його життєвого циклу. Загальноприйнятих назв для цих фаз і їх числа поки що ще немає. Але й особливих розбіжностей з цього питання немає. Тому існує декілька варіантів розбиття життєвого циклу програмного забезпечення на етапи.

За тривалістю життєвого циклу програмні продукти можна розділити на два класи: з малим і великим часом життя. Цим класам програм відповідають гнучкий (м'який) підхід до їх створення і використання і жорсткий промисловий підхід регламентованого проектування і експлуатації програмних виробів.

Програмні вироби з малою тривалістю експлуатації створюються в основному для рішення наукових та інженерних задач, для одержання конкретних результатів обчислень. Такі програми зазвичай відносно невеликі. Вони розробляються одним спеціалістом або маленькою групою. Головна ідея програми обговорюється одним програмістом і кінцевим користувачем. Деякі деталі заносяться на папір, і проект реалізується протягом декількох днів або тижнів. Вони не призначені для тиражування і передачі для подальшого використання в інші колективи. По суті, такі програми є частиною науково-дослідної роботи і не можуть розглядатися як відчужувані програмні вироби.

Їх життєвий цикл складається з тривалого інтервалу системного аналізу і формалізації проблеми, значного етапу проектування програм і відносно невеликого часу експлуатації і отримання результатів. Вимоги, що пред'являються до функціональних і конструктивних характеристик, як правило, не формалізуються, відсутні оформлені випробування програм. Показники їх якості контролюються тільки розробниками відповідно до їх неформальних уявлень [16].

Життєвий цикл програмного забезпечення (ЖЦПЗ) ділиться на шість фаз:

— аналіз вимог (АВ);
— проектування;
— реалізація (eng. coding);
— тестування та налагодження (eng. testing and debug);
— впровадження (eng. deployment);
— супровід (eng. support).

Аналіз вимог — це збір вимог до ПЗ, їх систематизація, виявлення суперечностей, відсутньої інформації і т. п. Аналіз вимог ділиться на три фази: збір, аналіз і документування.

Проектування — передбачає опис властивостей майбутньої системи, на основі аналізу вимог — результату попереднього етапу.

Реалізація — це безпосереднє кодування (або програмування) — процес написання програмного коду на певній мові програмування, з метою реалізації алгоритмів, визначених на попередньому етапі — проектуванні.

Тестування — перевірка і випробування закінченого продукту на предмет його якості: стійкості до навантажень, дружність до користувача (юзабіліті), безпеки (стійкості до зломів), відповідність вимогам і т. п.

Впровадження — це процес встановлення та налаштування програмного продукту для конкретних умов використання. Також, під впровадженням проводять навчання користувачів роботі з даним продуктом [4].

Супровід — процес підтримки програмного продукту. На даному етапі усуваються помилки («жучки»), вносяться зміни з метою поліпшити продукт. Ця стадія у життєвому циклі, як правило, займає більшу частину часу.

Життєвий цикл традиційно моделюється у вигляді деякого числа послідовних етапів або стадій, фаз. В даний час не вироблено загальноприйнятого поділу життєвого циклу програмної системи на етапи. Іноді етап виділяється як окремий пункт, іноді - входить в якості складової частини в більш великий етап. Можуть змінюватись дії, що виконуються на тому чи іншому етапі.

1.2 Моделі життєвого циклу програмного забезпечення

Життєвий цикл можна представити у вигляді моделей. В даний час найбільш поширеними є: каскадна, інкрементна (поетапна модель з проміжним контролем) і спіральна моделі життєвого циклу.

Каскадна модель (англ. waterfall model) — також відома як лінійно-послідовна модель життєвого циклу. Каскадна модель слідує в послідовному порядку, і тому ми переходимо до наступного кроку розробки або тестування, якщо попередній крок виконано успішно. Модель є дуже успішною для малих проектів, і якщо вимоги дуже чіткі. Тестування починається в кінці, коли робота з розробки завершена. Назва "водоспад" описує те, що тестування або розробка здійснюється вниз механізмом, як вода падає вниз. Після того, як в процесі розробки буде запущений будь-який наступний крок, ми не зможемо повернутися до попереднього кроку, щоб переробити або виконати будь-які зміни. Виправлення усвідомлених в процесі створення недоліків можливо, але вимагає додаткових угод до ТЗ.

Життєвий цикл традиційно поділяють на наступні основні етапи(рисунок 1.1):

— аналіз вимог;

— проектування;

— кодування (програмування);

— тестування та налагодження;

— експлуатація та супровід.

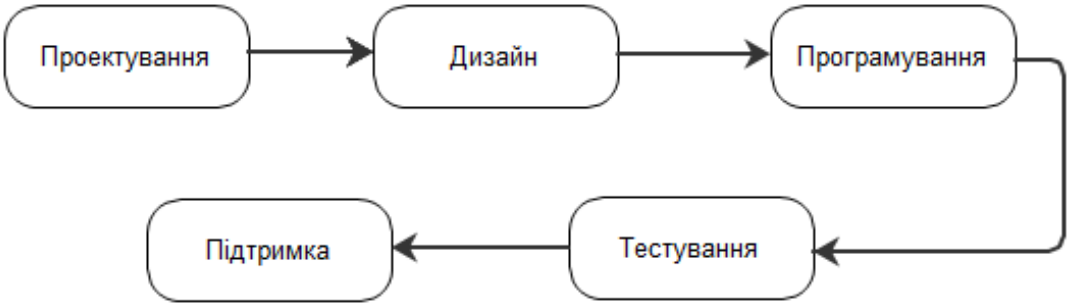

Рисунок 1.1 – Життєвий цикл каскадної моделі

Переваги моделі:

— стабільність вимог протягом всього життєвого циклу розробки;

— на кожному етапі формується закінчений набір проектної документації, що відповідає критеріям повноти і узгодженості;

— визначеність і зрозумілість кроків моделі і простота її застосування;

— виконувані в логічній послідовності етапи робіт дозволяють планувати терміни завершення всіх робіт і відповідні ресурси (фінансові, людські) [26].

Каскадна модель добре зарекомендувала себе при побудові відносно простих ПЗ, коли на самому початку розробки можна досить точно і повно сформулювати всі вимоги до продукту.

Недоліки моделі:

— складність управління проектом;

— істотна затримка отримання результатів;

— високий рівень ризику і ненадійність інвестицій;

— складність розпаралелювання робіт по проекту;

— надмірна інформаційна перенасиченість кожного з етапів; помилки і недоробки на будь-якому з етапів з'ясовуються, як правило, на наступних етапах робіт, що призводить до необхідності повернення на попередні стадії;

— кожна фаза є передумовою для виконання наступних дій, що перетворює такий метод у ризикований вибір для систем, що не мають аналогів, оскільки він не піддається гнучкому моделюванню [33].

Інкрементальна модель (англ. increment — збільшення, приріст) передбачає розробку програмного забезпечення з лінійною послідовністю стадій, але в кілька інкрементів (версій), тобто з запланованим поліпшенням продукту за весь час поки життєвий цикл розробки ПЗ не підійде до завершення [8].

Розробка програмного забезпечення ведеться по ітерації з циклами зворотного зв'язку між етапами. Між етапне коригування дозволяє враховувати реально існуючий взаємовплив результатів розробки на різних етапах, час життя кожного з етапів розтягується на весь період розробки(рисунок 1.2).

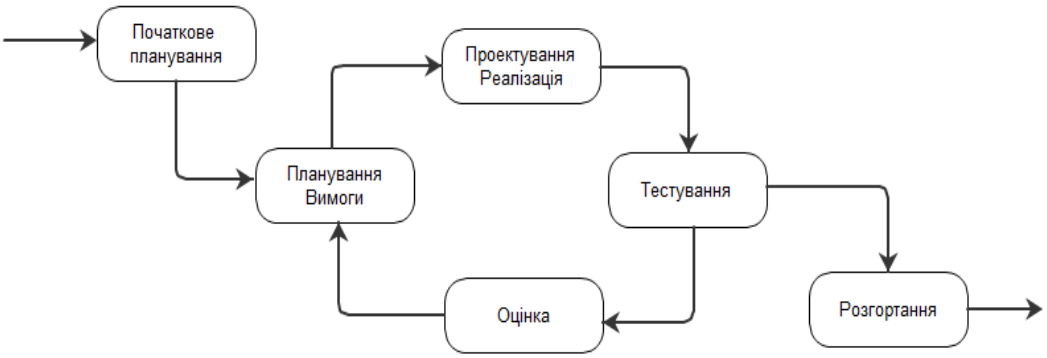

Рисунок 1.2 – Життєвий цикл інкрементальної моделі

На початку роботи над проектом визначаються всі основні вимоги до системи, поділяються на більш і менш важливі. Після чого виконується розробка системи за принципом збільшень, так, щоб розробник міг використовувати дані, отримані в ході розробки ПЗ. Кожен інкремент повинен додавати системі певну функціональність. При цьому випуск починають з компонентів з найвищим пріоритетом. Коли частини системи визначено, беруть першу частину і починають її деталізувати, використовуючи для цього найбільш відповідний процес.

У той же час можна уточнювати вимоги і для інших частин, які в поточній сукупності вимог даної роботи були заморожені. Якщо є необхідність, можна повернутися пізніше до цієї частини. Якщо частина готова, вона поставляється клієнту, який може використовувати її в роботі. Це дозволить клієнту уточнити вимоги для таких компонентів. Потім займаються розробкою наступної частини системи.

Ключові етапи цього процесу — проста реалізація підмножини вимог до програми та вдосконалення моделі в серії послідовних релізів до тих пір, поки не буде реалізовано у всій повноті [6].

Життєвий цикл даної моделі характерний при розробці складних і комплексних систем, для яких є чітке бачення (як з боку замовника, так і з боку розробника) того, яким має бути кінцевий результат. Розробка версіями ведеться в силу різного роду причин:

— відсутність у замовника можливості відразу профінансувати весь дорогий проект;

— відсутності у розробника необхідних ресурсів для реалізації складного проекту в стислі терміни;

— вимог поетапного впровадження та освоєння продукту кінцевими користувачами.

Впровадження всієї системи відразу може викликати у її користувачів неприйняття і тільки "загальмувати" процес переходу на нові технології. Образно кажучи, вони можуть просто "не переварити великий шматок, тому його треба подрібнити і давати по частинах".

Переваги і недоліки цієї моделі такі ж, як і у каскадної. Але на відміну від класичної стратегії замовник може раніше побачити результати. Вже за результатами розробки та впровадження першої версії він може незначно змінити вимоги до розробки, відмовитися від неї або запропонувати розробку більш досконалого продукту з укладенням нового договору [12].

Переваги моделі:

— витрати, які виходять у зв'язку зі зміною вимог користувачів, зменшуються, повторний аналіз і сукупність документації значно скорочуються порівняно з каскадною моделлю;

— легше отримати відгуки від клієнта про виконану роботу — клієнти можуть озвучити свої коментарі стосовно готових частин і можуть бачити, що вже зроблено. Так як перші частини системи є прототипом системи в цілому у клієнта є можливість швидко отримати і освоїти програмне забезпечення — клієнти можуть отримати реальні переваги від системи раніше, ніж це було б можливо з каскадною моделлю.

Недоліки моделі:

— менеджери повинні постійно вимірювати прогрес процесу, у разі швидкої розробки не варто створювати документи для кожної мінімальної зміни версії;

— структура системи має тенденцію до погіршення при додаванні нових компонентів — постійні зміни порушують структуру системи.

Щоб уникнути цього потрібен додатковий час і гроші на рефакторинг. Погана структура робить програмне забезпечення складним і дорогим для подальших змін. А перерваний життєвий цикл ПЗ призводить до великих втрат.

Схема не дозволяє оперативно враховувати нові зміни і уточнення вимог до ПЗ. Узгодження результатів розробки з користувачами проводиться тільки в точках, що плануються після завершення кожного етапу робіт, а загальні вимоги до ПЗ зафіксовані у вигляді технічного завдання на весь час її створення. Таким чином, користувачі часто отримують ПЗ, що не задовольняє їх реальним потребам [14].

RAD-модель — різновид інкрементальної моделі [3], яка фокусується на короткому часі розвитку. Ця модель є "швидкісною" моделлю, яка адаптує багато кроків від моделі каскадної, в якій швидке зростання досягається за допомогою компонентного підходу до розробки.

У випадку, якщо вимоги проекту добре зрозумілі, а обсяг проекту добре відомий, тоді процес RAD дозволяє команді розробників створити повністю функціональну систему, тобто програмний продукт протягом дуже короткого періоду, ледь не за дні.

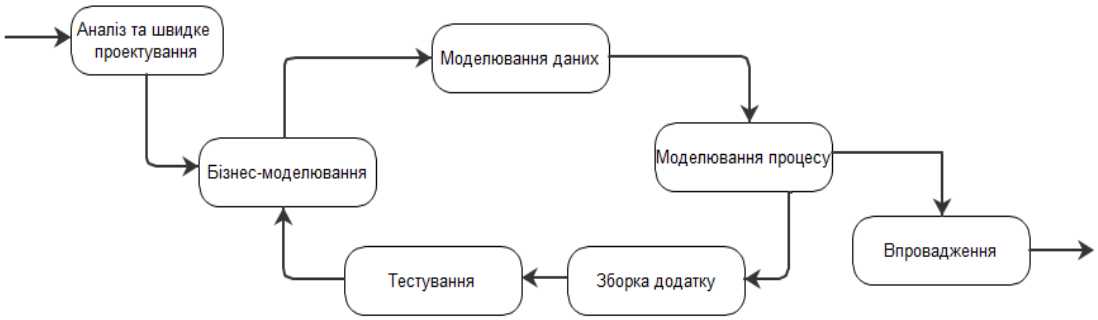

Рисунок 1.3 – Життєвий цикл RAD-моделі

Переваги моделі:

— швидка модель розробки додатків допомагає знизити ризик і необхідні зусилля зі сторони розробника програмного забезпечення;

— ця модель також допомагає клієнту робити часті перевірки («review») проекту;

— дана методика сприяє зворотньому зв'язку з клієнтами, яка завжди забезпечує обсяг поліпшень для будь-якого проекту розробки програмного забезпечення.

Недоліки моделі:

— ця модель залежить від сильної команди та індивідуальних мітингів для чіткого визначення вимог бізнесу;

— ця методологія працює лише на додатках, що можуть бути розбиті на модулі;

— такий підхід вимагає висококваліфікованих розробників і команди дизайнерів, що не під силу кожній організації.

Гнучка методологія використовується для проектування впорядкованого управління процесом розробки котрий дозволяє вносити постійні зміни в розробку проекту [4]. Ця методологія являється однією з концептуальних основ для створення різних проектів в галузі розробки програмного забезпечення. Ця модель використовується для максимального зменшення ризику при розробці продукту в короткі часові проміжки котрі називаються ітераціями і зазвичай тривають від одного тижня до одного місяця. Життєвий цикл методології наведено на рисунку 1.4.

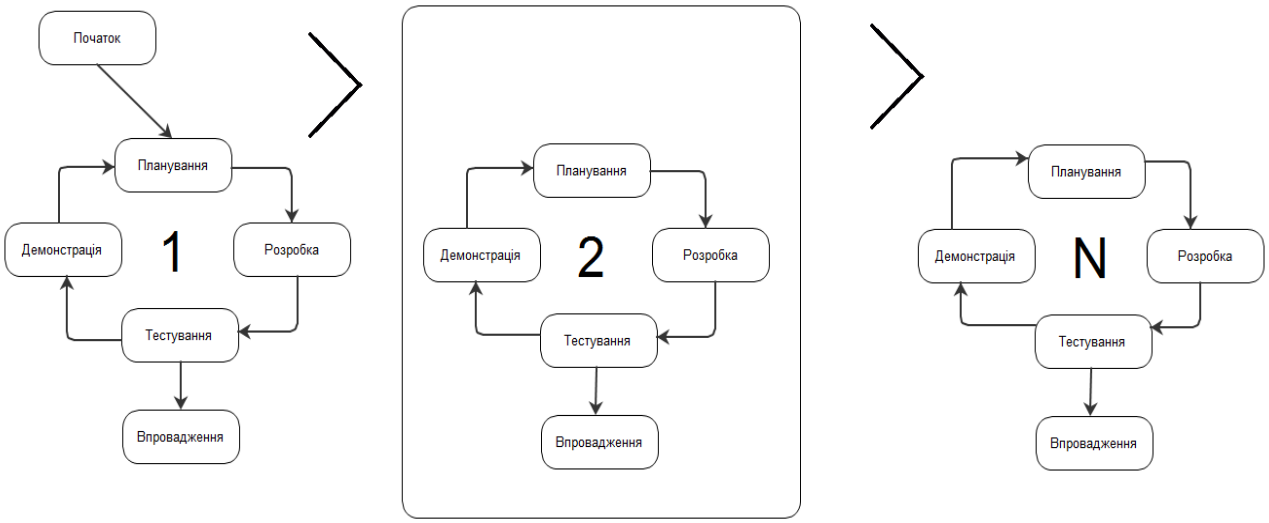

Рисунок 1.4 – Життєвий цикл гнучкої методології

Цю модель слід застосовувати коли потреби користувачів постійно змінюються в динамічному бізнесі. Зміни на Agile реалізуються за меншу ціну із-за постійних спринтів. На відміну від каскадної моделі, в гнучкій моделі для старту проекту досить лише невеликого планування.

Переваги моделі:

— гнучка методологія має адаптивний підхід котрий дозволяє змінювати вимоги клієнтів;

— безпосередній зв’язок та постійні відгуки замовників або їх представників не залишає місця для невизначеностей.

Недоліки моделі:

— ця методологія зосереджена на створенні програмного забезпечення раніше, ніж документації, звідси може бути нестача документації;

— процес розробки може вийти з під контролю, якщо замовник чітко не представляє кінцевий результат проекту [36].

Модель спірального життєвого циклу — це складна організація життєвого циклу ПО, яка фокусується на ранньому виявленні та зменшенні проектних ризиків. Розробка починається в невеликому масштабі, вирішуються локальні завдання, оцінюються ризики та шляхи їх зменшення. Наступний крок охоплює більш комплексні завдання - наступний виток спіралі.

Перевага підходу не в збільшенні швидкості розробки, а в зниженні рівня виникнення ризиків. Успішність спірального методу залежить від сумлінного, уважного і компетентного управління, а розмір проекту не має принципового значення.

Спіральна модель передбачає 4 етапи для кожного витка(рисунок 1.5):

— планування;

— аналіз ризиків;

— конструювання;

— оцінка результатів та перехід до нового витка спіралі.

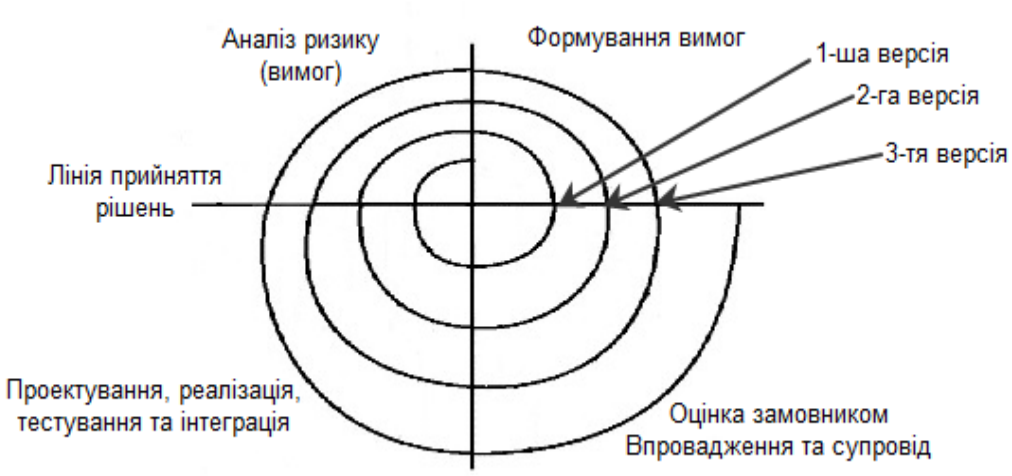

Рисунок 1.5 – Життєвий цикл спіральної методології

Переваги моделі:

— високий рівень аналізу ризиків;

— добре для великих і критично важливих проектів;

— сильне схвалення та контроль документації;

— додаткову функціональність можна додати пізніше;

— програмне забезпечення виробляється на початку життєвого циклу програмного забезпечення;

— зворотний зв'язок у напрямі від користувачів до розробників виконується з високою частотою і на ранніх етапах моделі, що забезпечує створення потрібного продукту високої якості.

Недоліки моделі:

— якщо проект має низький ступінь ризику або невеликі розміри, модель може виявитися дорогою. Оцінка ризиків після проходження кожної спіралі пов'язана з великими витратами.

— життєвий цикл моделі має складну структуру, тому може бути ускладнене її застосування розробниками, менеджерами і замовниками;

— спіраль може тривати до нескінченності, оскільки кожна відповідна реакція замовника на створену версію може породжувати новий цикл, що віддаляє закінчення роботи над проектом;

— велика кількість проміжних циклів може призвести до необхідності в обробці додаткової документації;

— використання моделі може виявитися дорогим і навіть неприпустимим за коштами, так як час витрачений на планування, повторне визначення цілей, виконання аналізу ризиків та прототипування, може бути надмірним;

— можуть виникнути труднощі при визначенні цілей та стадій, які вказують на готовність продовжувати процес розробки [11].

Застосування спіральної моделі доцільно в наступних випадках:

— при розробці проектів, що використовують нові технології;

— при розробці нової серії продуктів або систем;

— при розробці проектів з очікуваними істотними змінами або доповненнями вимог;

— для виконання довгострокових проектів;

— в проектах що вимагають демонстрації якості і версій системи або продукту через короткий період часу;

— при розробці проектів, для яких необхідний підрахунок витрат, пов'язаних з оцінкою і роздільною здатністю ризиків.

Раціональний уніфікований процес — одна з спіральних методології розробки програмного забезпечення [6]. В якості мови моделювання в загальній базі знань використовується мова Unified Modelling Language (UML).

Ітераційна розробка програмного забезпечення в RUP передбачає поділ проекту на кілька дрібних проектів, які виконуються послідовно, і кожна ітерація розробки чітко визначена набором цілей, які повинні бути досягнуті в кінці ітерації. Кінцева ітерація передбачає, що набір цілей ітерації повинен в точності збігатися з набором цілей, зазначених замовником продукту, тобто всі вимоги повинні бути виконані.

RUP досить добре формалізований, і найбільша увага приділяється початковим стадіям розробки проекту – аналізу й моделюванню. Таким чином, ця методологія спрямована на зниження комерційних ризиків (risk mitigating) за допомогою виявлення помилок на ранніх стадіях розробки. Технічні ризики (assesses) оцінюються і «розставляються» відповідно до пріоритетів на ранніх стадіях циклу розробки, а потім переглядаються з плином часу і з розвитком проекту протягом наступних ітерацій. Нові цілі з'являються в залежності від пріоритетів даних ризиків. Релізи версій розподіляються таким чином, що найбільш пріоритетні ризики усуваються першими [12].

Процес передбачає еволюцію моделей. Ітерація циклу розробки однозначно відповідає певній версії моделі програмного забезпечення. Кожна з ітерацій (workflow) містить елементи управління життєвим циклом програмного забезпечення (рисунок 1.6): аналіз і дизайн (моделювання), реалізація, інтегрування,

тестування, впровадження. У цьому сенсі RUP є реалізацією спіральної моделі, хоча досить часто зображується у вигляді графіка-таблиці.

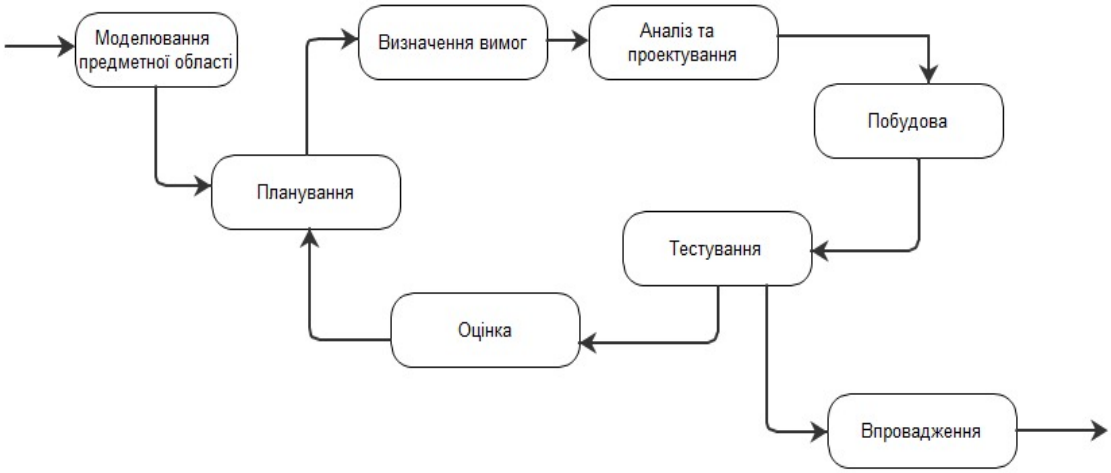

Рисунок 1.6 – Життєвий цикл RUP

У RUP рекомендовано дотримуватися шести практикам, що дозволяє успішно розробляти проект:

— ітеративна розробка;

— управління вимогами;

— використання модульних архітектур;

— візуальне моделювання;

— перевірка якості;

— відстеження змін.

Переваги моделі:

— ця методологія робить акцент на точній документації;

— це активно допомагає у вирішенні ризиків котрі пов’язані з зростаючими вимогами замовників та запитів керівництва;

— набагато зменшує потребу в інтеграції, оскільки процес інтеграції продовжується протягом всього процесу розробки.

Недоліки моделі:

— розробники повинні бути експертами свого діла в рамках цієї методології;

— процес розробки в цій методології дуже комплексний і не дуже точно організований;

— інтеграція під час усього процесу розробки додає проблем, що можуть викликати більше несправностей протягом стадії тестування;

— цей процес дуже комплексний, внаслідок цього він дуже важкий в розумінні.

Екстремальне програмування — гнучка методологія розробки програмного забезпечення [7]. Ця методологія, відома ще як «XP», в основному використовується для створення додатків в дуже нестабільному середовищі. Це додає великою гнучкості під час процесу моделювання. Головна перевага цієї методології – це те, що вона досить маловитратна. В моделі XP доволі часто буває, що вартість змін вимог на більш пізній стадії проекту може бути досить високою. Життєвий цикл XP наведено на рисунку 1.7.

Рисунок 1.7 – Життєвий цикл XP

Переваги:

— методологія екстремального програмування робить акцент на залученості замовника;

— ця модель допомагає визначити доцільні плани і графіки та дає можливість розробникам самостійно зафіксувати їхні графіки що є безумовно найбільшою перевагою;

— ця модель сумісна з більшістю сучасних методів розробки, що дає можливість розробникам робити якісний продукт.

Недоліки:

— ця методологія ефективна лише якщо розробники повністю зосереджені та захоплені розробкою;

— цей тип програмної розробки вимагає частих зустрічей, що дуже затратно для замовників;

— ця модель вимагає занадто багато змін в розробці, що дуже трудозатратно для розробника;

— в цій методології, як правило, неможливо визначити точні трудозатрати тому, що на початку проекту ніхто не уявляє цілий об’єм робіт та вимог [37].

Найбільш популярними та одними із батьків усіх інших є каскадна та Agile методології. З однієї сторони (каскадна) ми плануємо все до найменших деталей, визначаємо жорсткі терміни, маємо фіксований бюджет, але проблеми приходять коли потрібно зробити якісь правки або взагалі змінити русло куди йде розробка програмного продукту. З іншої сторони (Agile) маємо гнучку систему спринтів, замовник платить за спринти, досить легко вносити зміни, але досить складно встановлювати терміни для випуску продукту та важко прорахувати бюджет.

На рисунку 1.8 зображено координатну площину, на якій розташовано усі вище розглянуті методології. Вісь абсцис визначається як від «низькоформалізованих» до «високоформалізованих» методологій. Вісь ординат визначається як від «ітераційних» до «каскадних» методологій. Як видно з графіку маємо дві діаметрально протилежні методології це XP та каскадна, що є очевидним з вище сказаного. Оскільки, XP направлена на швидку розробку з невеликим ітераціями та

дозволяє досить легко вносити зміни, а каскадна, навпаки, потребує чітко визначених умов, має визначені етапи, які не дозволяють повернутися назад.

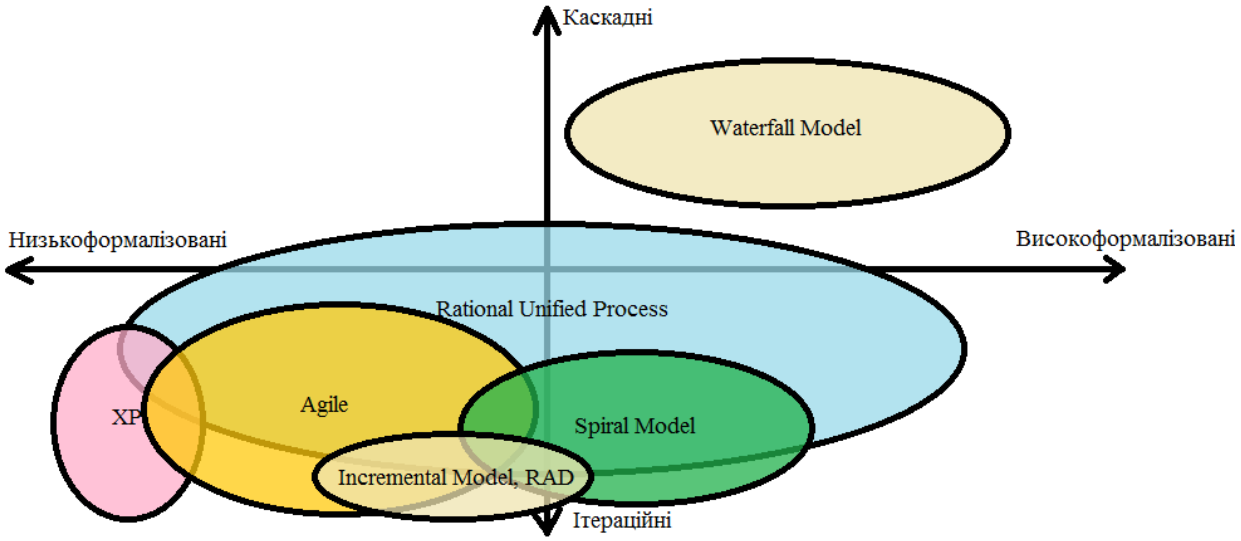

Рисунок 1.8 – Розташування методологій

У таблиці 1.1 проведено порівняння вищезгаданих методологій за певними критеріями. Як видно з таблиці, ітераційні методології мають такі переваги як гарантія успіху та менший рівень ризику, але мають менший контроль витрат. Каскадні ж мають досить високий контроль витрат, вищу детермінованість вимог, але повністю або частково не мають можливості повернення до попередньої фази.

Таблиця 1.1 — Порівняння методологій

	Каскадна	Інкрементальна	RAD	Agile	Спіральна	RUP	XP
Детермінованість вимог	повністю	основні	майже повн.	основні	бажано основні	майже повн.	початкові
Розмір проекту	нижче середн.	середній	малий	великий	великий	вище середн.	нижче середн.

Контроль витрат	високий	середній	вище середн.	низький	нижче середн.	вище середн.	низька
Гарантія успіху	невисока	вище середн.	середня	висока	висока	вище середн.	середня
Рівень ризику	високий	низький	нижче середн.	низький	низький	середній	вище середн.
Залученість замовника	низька	середня	середня	вище середн.	середня	нижче середн.	висока
Простота використання	висока	середня	низька	вище середн.	нижче середн.	нижче середн.	низька
Повернення до попередньої фази	–	–/+	+	+	–/+	–/+	+

1.3 Визначення вимог до програмних систем

Будь-яка програмна система створюється для вирішення однієї або декількох проблем майбутніх користувачів програмної системи. Програма – це ні що інше, як певний алгоритм, закладений в комп'ютер для вирішення певного кола завдань, робота якого повинна принести користувачеві відчутний результат.

Тут криється одна з проблем розробки програмного забезпечення (ПЗ). Програмісти і користувачі говорять на різних мовах. Програмісти знають як писати програми, а користувачі знають, або, принаймні, повинні знати, що повинна робити для них програма. Однак користувачі не ідеальне джерело інформації. Більшість користувачів знають, як виконувати свою роботу, проте далекі від поняття того, як перекласти все це на комп'ютер і часто не можуть викласти свої вимоги до майбутньої системи [4].

Традиційний підхід до вирішення цієї проблеми – це доручення визначення вимоги аналітикам, які проводять з користувачами інтерв'ю, виявляючи їх реальні потреби. Але навіть аналітикам важко отримати несуперечливий і надалі мало

змінний список вимог, якщо не використовувати систематизований підхід до визначення вимог.

Основна мета робочого процесу визначення вимог полягає в тому, щоб спрямувати процес розробки на отримання правильної системи. А правильна система – це система, яка робить те, що необхідно і нічого більше. Звичайно, програмістам важко робити систему так, щоб нічого від себе не докласти, і тим більше нічого не забути. Опис вимог повинно бути досить хорошим, для того щоб між користувачами та розробниками могло бути досягнуто розуміння того, що система повинна робити і чого не має. В іншому випадку користувачі будуть вважати, що система може зробити для них все, а програмісти не будуть розуміти, які функції майбутньої системи обов'язково повинні будуть включені в першу версію, і без них не можна обійтися, а які можна відкласти до наступного релізу.

Вимоги до програмного забезпечення — це те, що дана програма робить для користувача, приладу або іншої системи. Починати пошук слід серед того, що входить в систему і "виходить" з неї, тобто необхідно розглянути взаємодії системи з її користувачами. Для цього найпростіше спочатку уявити собі систему як якийсь чорний ящик і подумати про те, що слід визначити, щоб повністю описати, що робить цей чорний ящик. Крім вхідної та вихідної інформації, також необхідно звернути увагу на деякі інші характеристики системи, в тому числі на її продуктивність і інші типи складної поведінки, а також на інші способи взаємодії системи з її середовищем [1].

На рисунку 1.9 викладено представлення опису розробки вимог з бази знань SWEBOK.

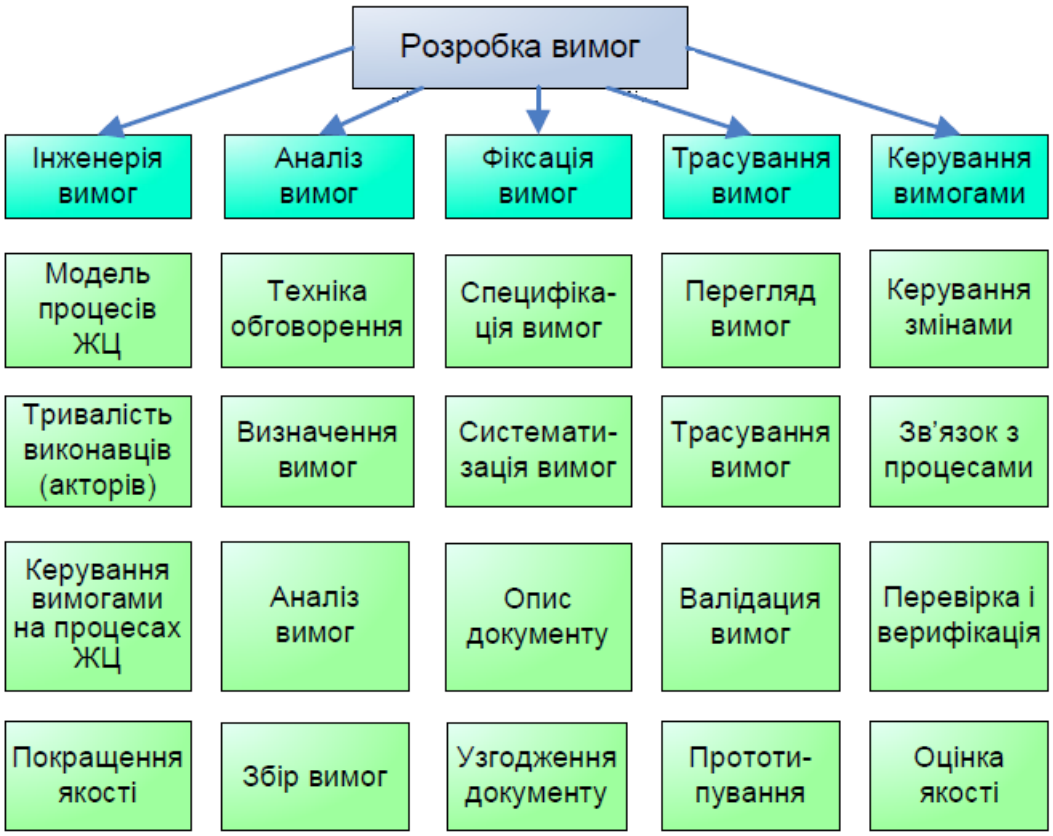

Рисунок 1.9 - Основні розділи розробки вимог

Використовуючи аналогічний підхід, Девіс зазначив, що для повного визначення системи необхідно описати наступні п'ять основних категорій елементів.

1. Вводи системи. Необхідно не тільки вказати на вміст введення, але і, якщо потрібно, докладно описати пристрій, а також протокол (форму, зовнішній вигляд і зміст введення). Як відомо більшості розробників, цей клас може містити значний обсяг відомостей і зазнавати частих змін, особливо в середовищах GUI, мультимедіа та Internet.

2. Виходи системи. Потрібно описати підтримувані пристрої виводу, такі як мовний висновок або відеотермінал, а також протокол і формати, що генеруються системою інформації.

3. Функції системи. Відображення вводів у висновки і їх різні комбінації.

4. Атрибути системи. Типові не поведінкові вимоги, такі як надійність, зручність супроводу, доступність і пропускна здатність, які повинні враховувати розробники.

5. Атрибути системного середовища. Це такі додаткові не поведінкові вимоги, як здатність системи функціонувати в умовах певних операційних обмежень і навантажень, а також сумісність з операційною системою [10].

Можна визначити наступні кроки робочого процесу визначення вимог:

— перерахування можливих вимог;

— усвідомлення контексту системи;

— визначення функціональних вимог;

— перерахування можливих вимог.

Перше, що потрібно зробити – це почати збирати всілякі вимоги та ідеї щодо майбутньої системи. Це буде несистематизований список, в який має потрапити все, що стосується системи і приходить в голову розробникам, аналітикам і користувачам. Ці ідеї будуть кандидатами на реалізацію в майбутніх версіях системи і використовуватимуться для планування робіт.

Кожне речення у списку повинно мати коротку назву та короткий опис, в чому воно полягає, також для подальшої роботи необхідна додаткова інформація для планування і подальшої реалізації вимог, в які можуть входити:

— стан пропозиції (наприклад, запропоновано, схвалено, включено в план, затверджено);

— трудомісткість в людино-годинах або вартість реалізації;

— пріоритет (наприклад, критичний, важливий або допоміжний);

— рівень ризику, пов'язаний з реалізацією пропозиції (наприклад, критичний, значний або звичайний) [26].

Цей список в ході робіт може зменшуватися, коли вимоги перетворюються в інші артефакти, наприклад – варіанти використання або нефункціональні вимоги, і збільшуватися, коли висуваються нові пропозиції.

Для того щоб вірно визначити вимоги розробники системи повинні розуміти контекст (частина предметної області) в якому працює система. Існує принаймні два підходи до опису контексту системи:

— моделювання предметної області;

— бізнес-моделювання.

Модель предметної області описує важливі поняття предметної області та їх зв'язок між собою. Не можна плутати модель предметної області з логічною та фізичною моделями системи. Модель предметної області описує тільки об'єкти предметної області, але не показує, як програмна система буде з ними працювати. Дана модель також дозволяє скласти глосарій системи для кращого її розуміння користувачами і розробниками.

Бізнес-модель описує процеси (існуючі або майбутні), які повинна підтримувати система. Бізнес-модель можна представити як підмножину моделі предметної області. Крім визначення бізнес-об'єктів, залучених у процес, ця модель визначає працівників, їх обов'язки, і дії, які вони повинні виконувати.

Підхід до виявлення системних вимог заснований на використанні варіантів використання системи (Use Cases), які охоплюють як функціональні, так і нефункціональні вимоги, які специфічні для конкретного варіанта використання.

Для користувача важливо, щоб система виконувала певні дії для нього, при цьому користувач певним чином взаємодіє з системою, використовує її для своїх цілей. Таким чином, якщо визначити всі можливі варіанти використання системи користувачем або іншими зовнішніми процесами, то ми отримаємо функціональні вимоги до неї.

Однак кожен конкретний користувач працює з системою по-своєму, тому для визначення дійсних варіантів використання системи необхідно досконально знати потреби всіх зацікавлених користувачів, провести аналіз отриманої інформації і систематизувати її в дійсні варіанти використання системи, тобто абстрагуватися від конкретних користувачів і виходити від бізнес-завдань.

На додаток до варіантів використання необхідно визначити, як повинен виглядати користувацький інтерфейс для реалізації того чи іншого варіанту використання. Накидати ескізи, обговорити їх з користувачами, створити прототипи і віддати їх на тестування користувачам.

До функціональних вимог належать такі властивості системи, як обмеження середовища і реалізації, продуктивність, залежність від платформи, розширюваність, надійність і т. д. Під надійністю розуміються такі характеристики, як придатність, точність, середнє напрацювання на відмову, число помилок на тисячу рядків програми, число помилок на клас.

Вимоги по продуктивності – це швидкість, пропускна здатність, час відгуку, використовувана пам'ять. Багато вимог, які пов'язані з продуктивністю, повинні бути описані в конкретних варіантах використання, а не в розділі, що відноситься до всієї системи.

Також можна відзначити, що часто нефункціональні вимоги не можуть бути прив'язані до конкретного варіанта використання і повинні бути винесені в окремий список додаткових вимог до системи.

1.4 Визначення поняття сервісу програмного забезпечення

Сервіс в контексті розробки ПЗ це лише компонент, елемент, складова частина програмного комплексу. За наявності більше ніж двох таких компонентів

Сервіс програмного забезпечення — це не є просто ще одним програмним продуктом, що є частиною якогось комплексу — це і є програмний комплекс який має за мету покращити існуючі процеси розробки ПЗ та витіснити класичні рішення своїм революційним підходом (додаток А).

СПЗ це комплекс який включатиме в своєму складі підсистеми необхідні для керування користувачами(як локальними так і хмарними), створення та ведення проектів, як програмних та і ні, взаємодії замовників та виконавців в межах проектів

і не тільки, підтримки життєвого циклу ПЗ, контроль розповсюдження та їх якість. А також можливість реалізувати багато інших підсистем для вирішення все більшої кількості необхідних задач.

Переваги, які надає використання концепції програмного сервісу(додаток А):

— середовище для спілкування з клієнтами щодо сервісу ПЗ, що надається;

— мінімізацію часу та зусиль на створення технічного завдання на розробку ПЗ, методики перевірок, тестувань та іншої супутньої документації;

— можливість формування поточних версій деяких паперових документів в будь-який час;

— можливість надгнучкої зміни вимог до ПЗ;

— систему встановлення пріоритетів реалізації вимог узгоджену з клієнтом;

— можливість та середовище обліку помилок та відстеження їх виправлення;

— спрощення розгортання та оновлення ПЗ;

— гнучкі механізми прогнозування та відстеження часових термінів реалізації вимог, задач, робіт, тощо;

— значне підвищення взаєморозуміння з клієнтом та максимальне уникнення конфліктних ситуацій;

1.5 Взаємодія клієнта та провайдера сервісу програмного забезпечення

Взаємодії замовника та виконавця, споживача та постачальника, абонента та провайдера завжди приділялося багато уваги. Оскільки взаємодія вищезазначених дійових осіб як правило вимагає тісного документального супроводження, було розроблено багато програмних систем для підтримки інформаційного середовища та обліку артефактів такої взаємодії. Особливого значення такі системи набули в області розробки програмного забезпечення, де взаємодія замовника і виконавця

часто відіграє ключову роль в успіхах або провалах при реалізації проектів пов’язаних з інформаційними технологіями.

Проблема інформаційної підтримки взаємодії сторін виходить на новий рівень при впровадженні концепції надання сервісу програмного забезпечення (надалі СПЗ), що передбачає перехід від продуктового підходу до розробки програмного забезпечення до сервісного [21].

Специфіка взаємодії сторін в разі використання СПЗ полягає в тому, що:

– не існує конкретного технічного завдання або його аналогу на момент укладання договору про співпрацю;

– вимоги до ПЗ, що розробляється, породжуються ітераційно, починаючи з однієї початкової вимоги – задовольнити клієнта;

– вимоги до ПЗ змінюються (об’єднуючись, розгалужуючись, зникаючи та перетинаючись) навіть під час або після їх реалізації;

– не існує чітких віх фіксації результату розробки (немає ні версій, ні етапів, ні визначених наперед кроків);

– функціонал поставляється замовнику безперервно по мірі готовності по принципу безперервної інтеграції;

– оплата виконаних робіт передбачає (необов’язково) передплатні внески (prepaid);

– облік оплати виконаних робіт не оговорюється заздалегідь, а відбувається «по факту» за кожну вимогу у вигляді фіксації автоматичного списання умовних коштів;

– оцінка реалізації вимог в грошовому еквіваленті відбувається спільно представниками клієнта та провайдера в режимі «аукціону»;

– існує можливість одностороннього залучення до взаємодії третіх сторін (часто це контролюючі, перевіряючі або наглядаючі ролі при взаємодії);

– можливість заміни виконавця або замовника СПЗ без втрати історії попередньої взаємодії. [29]

Для врахування зазначених особливостей СПЗ в системі «CleanSlate» (надалі, CS) реалізується комплексна підсистема управління вимогами – «Аукціон вимог».

Під «Аукціоном вимог» слід розуміти частину функціоналу системи, що відповідає за процес переговорів та затвердження умов (суть, терміни виконання, вартість) вимоги між сторонами замовника та розробника. Головна ідея полягає у тому, що замовник та менеджер введуть перемовини про характеристики вимоги, щоразу її деталізуючи та уточнюючи. В рекомендованому розвитку подій (згідно з концепцією СПЗ) вимога береться до опрацювання лише коли має остаточне двостороннє погодження. Розглянемо на прикладі. Замовник створює вимогу «Розробити систему для інтеграції з хмарними обчисленнями» та виставляє початкову ціну в 10000 у.о. з термінами виконання основного функціоналу до 30.12, про це сповіщається менеджер проекту, що не згоден з ціною, він її підвищує до 15000 у.о. та обов’язково додає коментар-пояснення, наприклад, «Необхідні додаткові розробники». Далі система сповіщає про це замовника, а він в свою чергу вирішує, що на додаткового розробника достатньо замість 5000 у.о. всього 2000 у.о. Тобто знижує ціну до 12000 у.о. і продовжує терміни виконання до 07.01 та дає зі своєї сторони згоду. Менеджер бачить зміни в ціні та термінах, приймає рішення, що це його влаштовує та дає зі своєї сторони згоду. Після цього вимога буде затверджена і перейде у стадію виконання, що супроводжується створенням задач (описано нижче).

Типовий життєвий цикл вимоги складається з наступних етапів: створена («To Discuss») – обговорюється («In Discuss») – обговорення завершено («Accepted») – почалася розробка («In Progress») – завершено виконання вимоги («Completed»). Також передбачається два додаткових етапи перехід до яких можливий із етапів In Discuss, Accepted, In Progress: обговорення чи виконання відкладено до невизначених термінів («Frozen») та обговорення чи виконання відмінено («Canceled»).

Для роботи з вимогами та підвимогами передбачено дві додаткові дії такі як розгалуження («Branching») та злиття («Merging»). Під розгалуженням слід розуміти

процес декомпозиції вимоги/підвимоги на дві або більше. Під злиттям слід розуміти зворотній процес до розгалуження – об’єднання двох або більше вимог чи підвимог. Концепція СПЗ рекомендує виконувати дані дії до етапу Accepted, оскільки виконання даних операцій після даного етапу потребує додаткових обговорень та можливо витрат і зміщень кінцевих термінів [6]

Варто відмітити те, що після надання двосторонньої згоди («Accepted») й переходу безпосередньо до розробки («In Progress»), грошові кошти вже починають поступати до провайдера (списуватися з балансу рахунку в межах системи, необов'язково напряму з банківського рахунку). У випадку переходу до стану Canceled, наприклад, сторона виконавців все одно отримає гроші в межах затрачених людино/годин. Вимога в свою чергу, після досягнення двосторонньої згоди, декомпозується на задачі, ті в свою чергу рекурсивно можуть розпадатися на підзадачі. Кожна задача має свій процент виконання чи іншу умовну одиницю для оцінки завершеності задачі (стану). Усі стани агрегуються з урахуванням вагових коефіцієнтів, що були назначені відповдним учасником сторони провайдера.

Для замовника усі розбиття вимоги на задачі й підзадачі невидимі – він не бачить ні їх проценту виконання, ні визначених термінів і т.д. Він лише бачить агрегований стан виконання вимоги, що утворився із відповідних показників для задач.

Отже, новий підхід до розробки ПЗ як надання сервісу пропонує надгнучкий інструментарій для управлінням проектом. Він передбачає усі необхідні стани для вимог, що є основою комунікації між сторонами замовника й провайдера. Сама ж взаємодія для затвердження вимоги висвітлена як «Аукціон вимог», який надає можливість дійти згоди деталізуючи й уточнюючи вимоги.

1.6 Особливості проектування програмного забезпечення

Проектуванням програмного забезпечення є процесом створення проекту програмного забезпечення (ПЗ). Проектування програмного забезпечення являє собою окремий випадок проектування процесів і продуктів. Мета проектування передбачає визначення внутрішніх властивостей системи і деталізацію її зовнішніх (видимих) властивостей у відповідності з виданими замовником вимог до програмного забезпечення (вихідними умовами задачі), які, в свою чергу, піддаються аналізу.

Хід процесу проектування ПЗ і його результати будуть залежати не тільки від складу вимог, але і від досвіду проектувальника (розробника) і від обраної моделі процесу проектування. Після визначення вимог до програмного забезпечення розробником будуть отримані узгоджений чіткий план дій, графік строків і оплат. У той же час розробник може скоротити час розробки і підвищити її якість, а також дозволяє передбачити будь-які інші нюанси розробки, наприклад, юридичні (передача авторських прав на проектоване програмне забезпечення), дій, графік строків і оплат [9].

При проектуванні ПЗ заздалегідь розробник має можливість: оцінити час розробки і вартість програмного продукту; виключити втрати матеріальних витрат та часу на вимушені доопрацювання, непотрібні дії, тривале узгодження; уникнути незадоволеності і розбіжностей між замовником і виконавцем. Порядок розробки програмного забезпечення в залежності від особливостей проекту може відрізнятися, але в загальному вигляді він складається з наступних етапів:

— підготовки;
— проектування;
— створення, що включає дизайн, кодування, тестування, документування;
— підтримки, що включає впровадження та супровід.

У процесі підготовки до проектування повинні бути вирішені організаційні питання: необхідно визначити склад робіт, а для цього потрібно дізнатися, що може надати замовник (технічне завдання, дизайн, макети), достатні вихідні коди і наскільки, які етапи вони закривають. Визначитися з бюджетом і термінами: на

підставі наявних матеріалів стверджують приблизну вартість, загальний термін всього проекту, а також строк та точну вартість найближчого етапу. Після вирішення організаційних питань підписують контракт, отримують передоплату і необхідні для роботи матеріали.

Проектування складається з наступних етапів:

— описи — даний етап включає в себе спільну роботу замовника (визначає користь продукту, вимоги до зовнішнього вигляду і працездатності) і розробника (пропонує алгоритмічні та технічні рішення поставленої задачі);

— визначення архітектури — на даному етапі стверджується мова програмування, бази даних і фреймворки сервери;

— розробка технічного завдання (ТЗ) — ТЗ становить архітектор згідно з описом і відповідями на питання замовника. Потім ТЗ погоджують з менеджером проекту, далі передають клієнту і виробляють правки;

— етапи розробки макетів, які потім додаються до ТЗ — на даному етапі розробляють макети принципових схем пристрою, інтерфейсів, діаграм структури бази даних, схем взаємодії компонентів;

— контролю — в ході цього етапу архітектором усуваються зауваження менеджера проектів;

— затвердження — на даному етапі замовником перевіряється і змінюється самостійно ТЗ, або повідомляється список правок проект-менеджеру. Після усунення зауважень ТЗ стверджують і докладають до контракту.

В результаті проектування виходить технічне завдання з однозначною і зрозумілою як для замовника, так і для виконавця(в якості виконавця можуть виступити керівник проекту, програмісти, тестувальники, дизайнери та інші учасники процесу розробки) постановкою вимог до ПЗ.

2 ПРОЕКТУВАННЯ СИСТЕМИ ПІДТРИМКИ ЖИТТЄВОГО ЦИКЛУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Архітектура системи

Оскільки система має досить багато модулів, то ж використаємо одразу декілька типів архітектур таких як клієнт-серверна, тришарова та мікросервісна, що взаємодіють за допомогою шини повідомлень [9-10].

На рисунку 2.1 зображено загальну архітектуру системи.

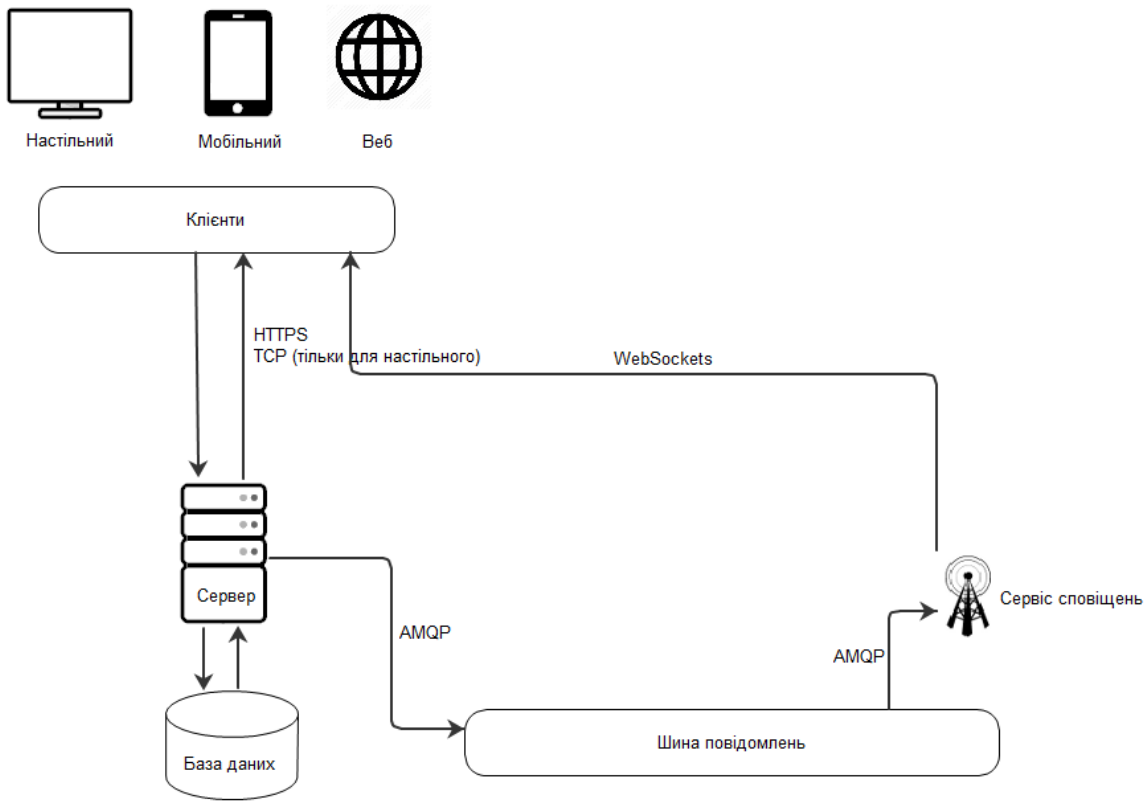

Рисунок 2.1 – Загальна архітектура системи

Як видно з рисунку 2.1 маємо дві основні взаємодіючи сторони: клієнт та сервер. Серверна частина системи забезпечує обробку запитів від клієнта, обробку

та зберігання даних, взаємодію з іншими модулями. Клієнтська частина займається здійсненням запитів до серверу та відображенням даних, що отримані від серверу. Також серверна частина взаємодіє з іншими сервісами, наприклад, сервіс сповіщень.

Для взаємодії клієнту та серверу було обрано два протоколи HTTPS та TCP. Вибір двох протоколів обґрунтовується тим, що наша система має дві версії: публічну та корпоративну.

Публічна версія – це сервіс за допомогою якого може будь-хто створювати проекти і запрошувати до участі будь-яких користувачів. Даний сервіс розгортається на наших потужностях.

Корпоративна версія – це той ж самий сервіс, що і публічний, але за однією головною відмінністю — він розгортається у корпоративній мережі. Це обґрунтовується тим, що клієнти котрі хочуть використовувати наш продукт досить сильно турбуются про безпеку своїх даних і хочуть їх зберігати лише на своїх потужностях, а також обмежити коло доступу користувачів [11].

Для публічної версії було обрано протокол HTTPS. Даний протокол дозволяє досить легко описувати публічний доступ до даних з використанням Web Api у стилі REST-архітектури. Нижче буде більш детально розписано.

Для корпоративної версії можна використовувати як HTTPS так і TCP. TCP не дає такою гнучкості як HTTPS, але дає більшу швидкість.

Архітектура сервера виконана за тришаровою архітектурою (рисунок 2.2).

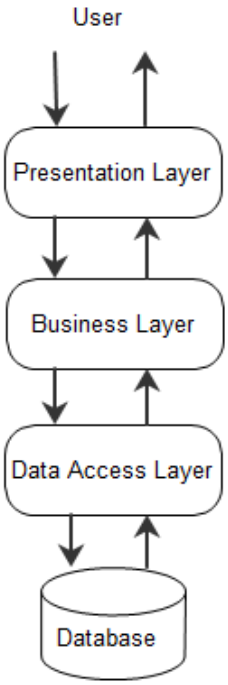

Рисунок 2.2 – Тришарова архітектура

Ця архітектура складається з трьох основних шарів:

– шар доступу до даних (Data Access Layer, DAL) – на цьому рівні виконується отримання та збереження даних. Роль провайдера даних можуть виконувати реляційні та нереляційні бази даних, веб-сервіси та інші;

– шар бізнес-логіки (Business Logic Layer, BLL) – на цьому рівні виконується обробка даних отримана з рівню доступу до даних та передача до рівню презентації та навпаки. На цьому рівні також виконуються перевірки на бізнес-правила;

– шар презентації (Presentation Layer) – на цьому рівні виконується отримання даних з нижчих рівнів та від користувача. Дані можуть представлятися у вигляді настільного додатку, веб додатку, Web Api та інші.

В нашій системі для збереження даних використовується реляційна база даних, відповідно, для рівня доступу до даних використовується відповідний провайдер БД.

Рівень представлення даних реалізує Web Api — для публічної версії, що також може використовуватися і в корпоративній та RPC (Remote Procedure Call) — тільки

для корпоративної. Завдяки використанню Web Api, що виконаний у стилі REST сервісу, ми отримаємо можливість написання клієнтів під будь-які платформи. Для REST використовуємо формат даних JSON, а для RPC – XML.

Клієнт як і сервер виконано за тришаровою архітектурою. Шар доступу до даних виконує REST-клієнт, що взаємодіє з Web Api, що надає наш сервер по протоколу HTTPS та RPC-клієнт, що взаємодіє з RPC сервера по протоколу TCP [14].

Шар презентації для публічної версії завдяки Web Api може бути виконано як настільний, мобільний та веб додаток. Для корпоративної – настільний додаток.

Оскільки система досить містить багато модулів, то було прийнято рішення для певних видів задач використовувати архітектуру мікросервісів. Мікросервісна архітектура – це підхід до створення додатків, що пропонує відмову від єдиної, монолітної структури (рисунок 2.3).

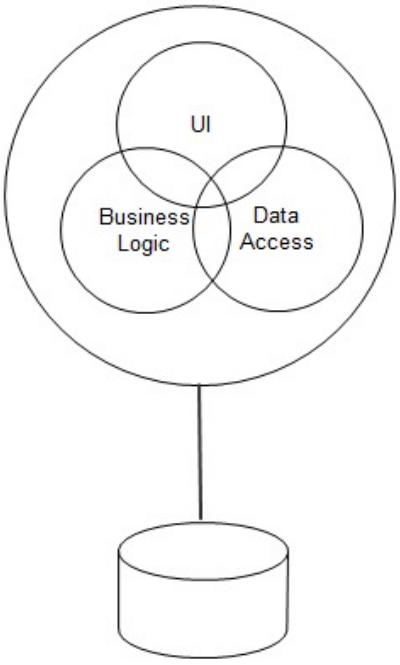

Рисунок 2.3 – Монолітна архітектура

Тобто замість того, щоб виконувати усі обмежені контексти додатку на сервері, ми використовуємо декілька невеликих додатків (рисунок 2.4), кожний з яких

відповідає деякому обмеженому контексту. При цьому ці додатки працюють на різних серверах та взаємодіють між собою по мережі.

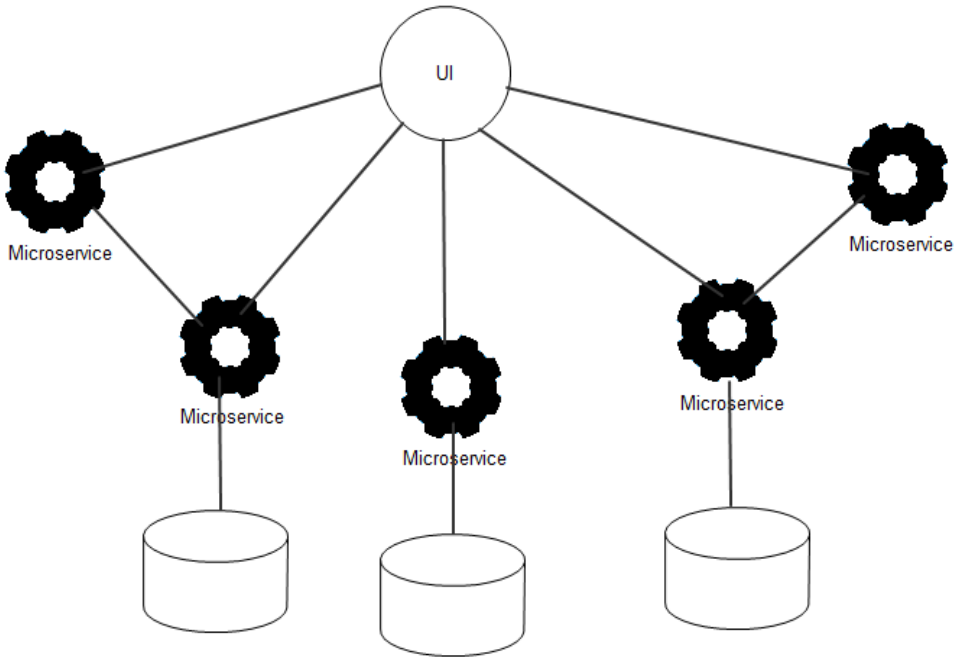

Рисунок 2.4 – Архітектура мікросервісів

В нашому випадку додаток, що має обмежений контекст – це сервіс сповіщень. Його головна задача повідомляти користувача у режимі реального часу про події, котрі відбулися і потребують того, щоб користувач їх побачив з мінімальною затримкою у часі.

Існують різні підходи для доставки повідомлень до користувача такі як pooling, long pooling, SSE та за допомогою WebSockets.

Pooling підхід – при цьому підході клієнт з деякою постійною затримкою у часі звертається до серверу, щоб дізнатися чи не з’явилися для нього нові повідомлення.

Long pooling підхід – працює за таким же принципом як pooling, але за однією відмінністю – при надсиланні запиту, сервер не віддасть відповідь поки не з’являться повідомлення для цього клієнту або поки не закінчиться максимальний час очікування.

SSE (Server-Sent Events) підхід – при цьому підході клієнт встановлює з сервером постійне однонаправлене з’єднання (від сервера до клієнта) і при необхідності сервер передає повідомлення до клієнта [34].

З використанням WebSockets – при цьому підході клієнт встановлює з сервером постійне двонаправлене з’єднання і при необхідності сервер відправляє повідомлення.

Для доставки повідомлень до клієнта ми обрали WebSockets. Хоча на даний момент нам потрібно лише відправляти сповіщення від сервера до клієнта, але щоб необмежувати свої можливості при використанні однонаправленого з'єднання, обрано двонаправлений. Наприклад, щоб можна було у майбутньому отримувати підтвердження про доставку повідомлення.

Для організації взаємодії між мікросервісами було обрано протокол AMQP [11]. AMQP (Advanced Message Queueing Protocol) – це публічний протокол для передачі повідомлень між компонентами системами. Головна ідея полягає в тому, що окремі підсистеми (чи незалежні додатки) можуть обмінюватися довільним чином повідомеленнями через AMQP-брокер, котрий виконує маршрутизацію, можливо гарантує доставку, розподіл потоків даних, підписку на необхідні типи повідомлень.

AMQP містить три основні поняття:

– повідомлення (message) – одиниця даних, які надсилаються, головна його частина (зміст) ніяк не інтерпретується сервером, до повідомлення можна додавати структуровані заголовки;

– точка обміну (exchange) – до неї поступають повідомлення. Точка обміну розподіляє повідомлення в одну чи декілька черг. При цьому в точці обміну повідомлення не зберігаються;

– черга (queue) — тут зберігаються повідомлення, доки вони не будь забрані клієнтом.

Як видно з рисунку 2.5 даний протокол має дві взаємодіючі сторони: видавець (publisher), той хто створює повідомлення і передає до точки обміну та споживач (consumer), той хто забирає повідомлення з черги та обробляє їх.

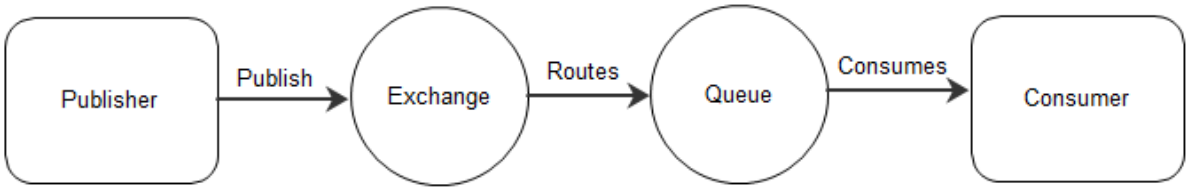

Рисунок 2.5 – Головні компоненти AMQP протокола

В нашій системі (згідно з архітектурою, що зображена на рисунку 2.1) у ролі видавця виступає «Сервер» - створює повідомлення та надсилає до точки обміну, і один з можливих споживачі – це сервіс сповіщень, що обробляє повідомлення, котрі пов’язання з сповіщенням користувача про деякі події.

Для забезпечення слабозв’язності модулів, їх легкої інтеграції та написання модульних та інтеграційних тестів було застосовано один із приципів ООП – інверсія управління.

Інверсія управління (Inversion of Control, IoC) – важливий принцип об’єктно-орієнтованого програмування, що використовується для послаблення зв’язності модулів [12]. Також архітектурне рішення інтеграції, що полегшує розширення можливостей системи, при якому контроль над потоком управління програми залишається за каркасом.

Однією з реалізацій IoC в застосуванні до управління залежностями є впровадження залежностей.

Впровадження залежностей (Dependency Injection, DI) – процес представлення зовнішньої залежності програмному компоненту. Впровадження залежностей використовується в багатьох фреймворках, котрі називаються IoC-контейнерами.

2.2 Система динамічних ролей та АРМ-ів

Дана система побудована таки чином, що користувач має можливість сам створювати нові ролі та надавати їм необхідних функцій та прав доступу. Щоб

спростити користування та заощадити час користувачу, в системі передбачено шість «типових» ролей: замовник (customer), сервіс менеджер (service manager, SM), менеджер проекта (project manager, PM), керівник команди (team lead, TL), розробник (developer), тестувальник (quality assurance engineer, QA).

Розглянемо кожну роль детально:

– замовник – до його основних функцій відносяться такі як створення вимог до розроблюваємого програмного продукту, виставлення бажаної ціни та кінцевих сроків;

– менеджер проекту – до його основних функцій відносяться такі як узгодження ціни та кінцевих сроків, шо виставлені для вимог, створення задач, що покривають ті чи інші вимоги, назначення відповідальних керівників команд на задачі та їх вчасного виконання;

– сервіс менеджер – до його основних функцій відносяться такі як надання сервісу незалежним замовникам, пошук виконавців, створення ролей, контроль робочого процесу;

– керівник команди – до його основних функцій відносяться такі як розбиття задач на підзадачі, назначення відповідальних розробників за підзадачу, звітування про процес виконання задач;

– розробник – до його основних функцій відносяться такі як вчасне та якісне виконання задачі, звітування про процес виконання підзадачі;

– тестувальник – до його основних функцій відносяться такі як затвердження про можливість використання нового функціоналу у програмному продукті, виконаня планових тестів (стрес-тестів, системних тестів і т.д.) та пошук недоліків [35]

Кожна з цих ролей потребує того чи іншого візуального супроводження. Одні ролі потребують певних візуальних компонентів котрі не потрібні або до них немає прав доступу інши та навпаки. Для досягнення даної задачі було обрано використання APM.

Як згадувалося вище у розділі 2.1.5, ми використовуємо при написанні модулів прицип інверсії управління, що забезпечує слабозв’язність. Завдяки цьому ми можемо динамічно завантажувати той чи інший АРМ у відповідності до ролі та прав користувача.

Нижче буде наведено основні АРМ.

За діаграмою прецендентів, що наведена на рисунку 2.6 можна виділити основні можливості замовника такі як створення та робота з вимогами, виставлення термінів виконання та узгодження ціни.

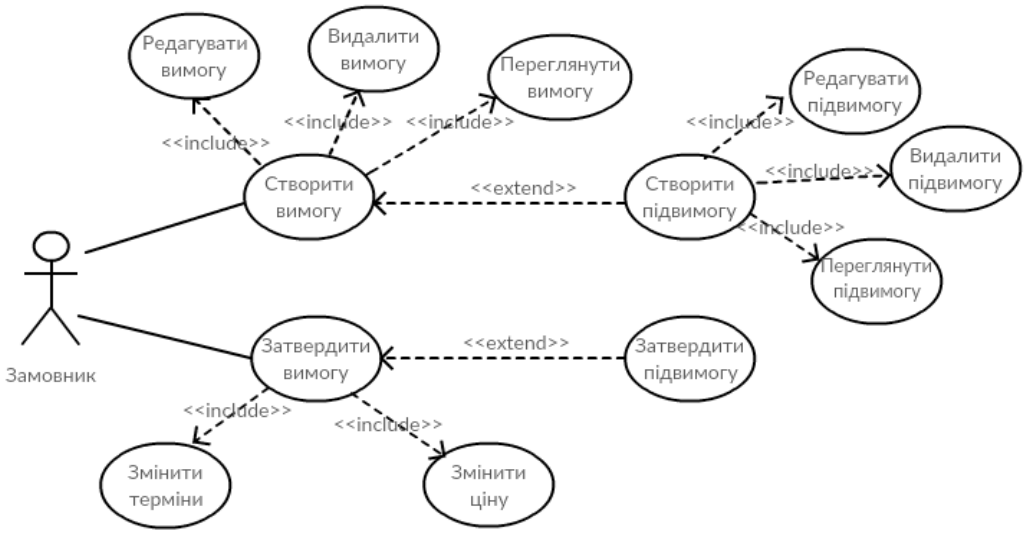

Рисунок 2.6 – Діаграма прецендентів для роботи з вимогами по відношенню до замовника

АРМ замовника використовує динамічний модуль для роботи з вимогами у повному обсязі, що має наступні функції: створювати, редагувати та видаляти вимоги, виставляти кінцеві сроки реалізації, вести перемови про ціну, терміни виконання та умови, підтверджувати вимогу зі сторони замовника.

За діаграмою прецендентів, що наведена на рисунку 2.7 можна виділити основні можливості менеджера проекту такі як перегляд існуючих вимог, створення задач та призначення керівників команд на задачі.

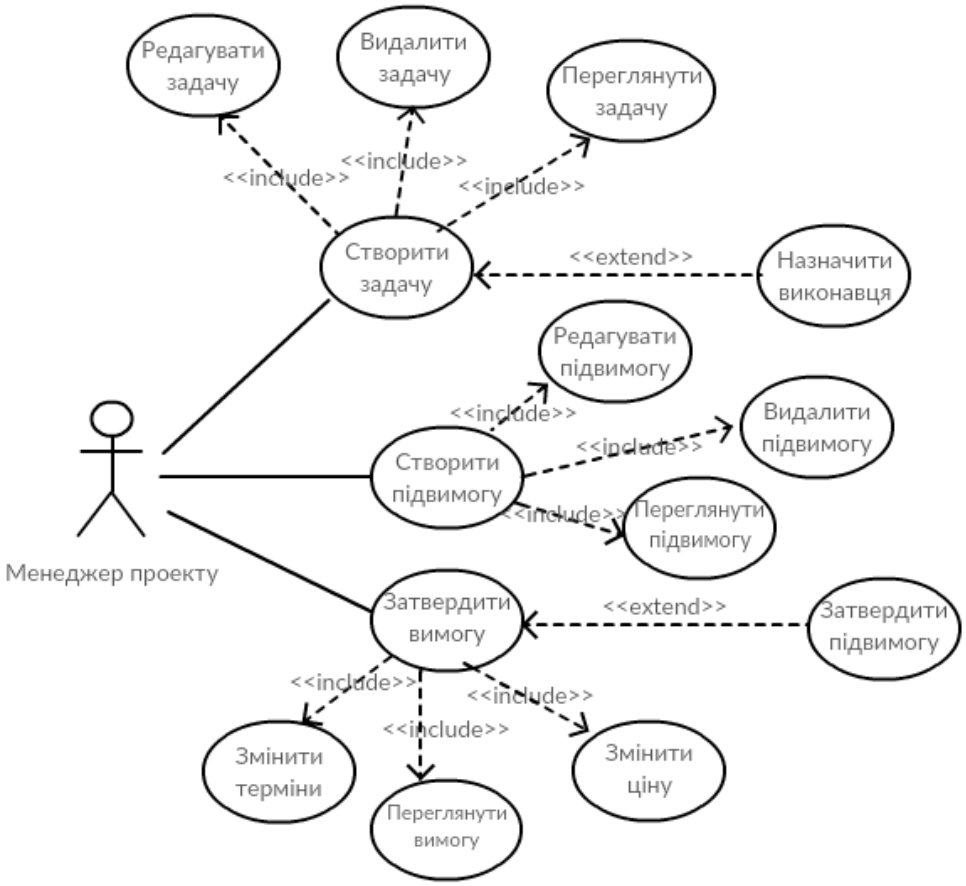

Рисунок 2.7 – Діаграма прецендентів для роботи з вимогами та задачами по відношенню до менеджера проекту

АРМ менеджера використовує динамічний модуль для роботи з вимогами з правами перегляду існуючих вимог, редагування ціни, надання підтвердження про ціну та динамічний модуль для роботи з задачами, що дозволяє створювати, редагувати та видаляти задачі, назначення зв’язків між задачами та вимогами, закріплення задач за керівниками команд, перегляд проценту виконання задачі, назначення кінцевих термінів виконання задачі, що не суперечить кінцевим термінами виконання вимог.

За діаграмою прецендентів, що наведена на рисунку 2.8 можна виділити можливості потреби керівника команди такі як перегляд існуючих задач, створення підзадач та призначення розробників на підзадачі.

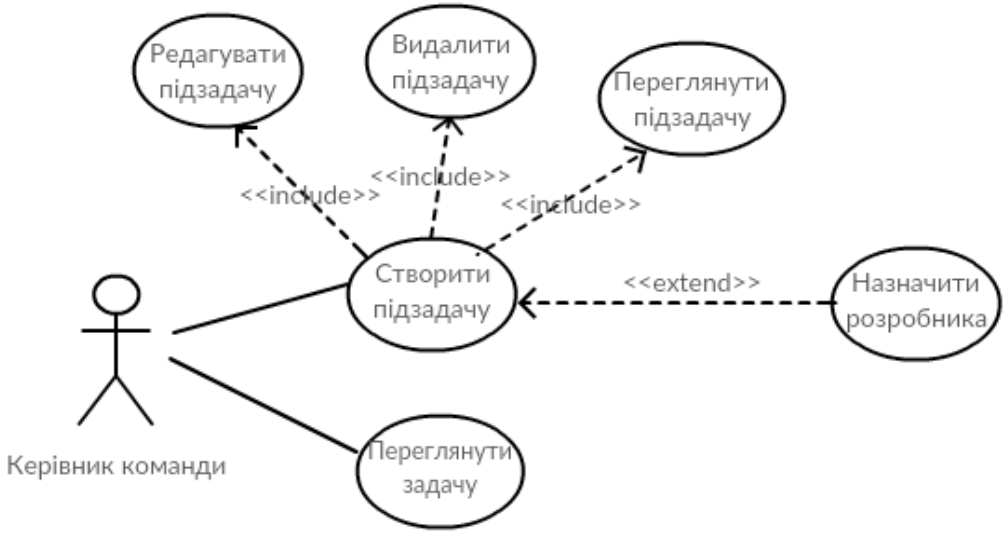

Рисунок 2.8 – Діаграма прецендентів для роботи з задачами та підзадачами по відношенню до керівника команди

АРМ керівника використовує динамічний модуль для роботи з задачами з правами перегляду існуючих назначених задач, створення, редагування, видалення підзадач до задачі, назначення кінцевих термінів виконання підзадач, що не суперечить кінцевим срокам задачі, назначення розробників до підзадач.

За діаграмою прецендентів, що наведена на рисунку 2.9 можна виділити основні можливості розробника такі як перегляд назначених підзадач та звітування про процент виконання підзадачі.

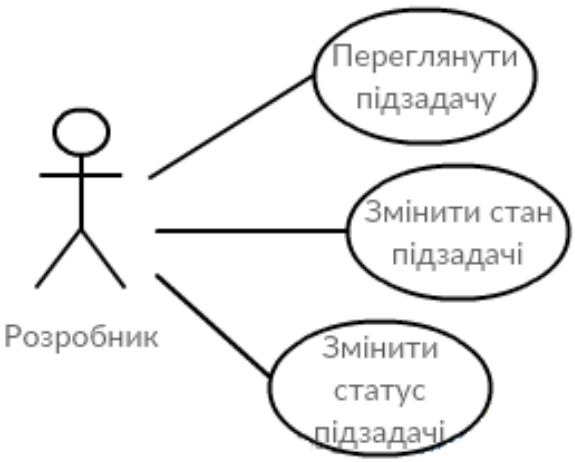

Рисунок 2.9 – Діаграма прецедентів для роботи з підзадачами по відношенню до розробника

АРМ розробника використовує динамічний модуль для роботи з задачами з правами перегляду назначених підзадач та їх задач, звітування про процент виконання підзадачі, відмічанням задачі як виконаної.

2.3 Основні функції системи

До основних можливостей підсистеми роботи з вимогами відносяться такі функції як: створення, редагування, видалення вимог та підвимог, міграція із записок до вимог або підвимог, сповіщення про стан виконання вимоги або підвимоги (під станом вимоги або підвимоги слід розуміти процент закінченості тієї або іншої).

Можливості, що надаються для роботи з вимогам та підвимогами згідно з обраною роллю (типова) описані у розділі 2.2. Для того, щоб конкретизувати вимогу передбачена можливість її логічної декомпозиції на підвимоги. Дане розбиття може виконувати як замовник так і керівник проекту .

На кожну підвимогу менеджером проекту створюється набір задач (одна чи більше). На вимогу також можна створювати задачі, але не обов’язково. Стан вимоги чи підвимоги може змінюватися автоматично (з урахуванням вагових коефіцієнтів) або вручну.

При автоматичній зміні стан вимоги розраховується з урахуванням вагових коефіцієнтів підвимог та задач, що задаються при створенні тих чи інших. Розглянемо на прикладі. Маємо вимогу, що складається з трьох підвимог (розрахунок стану підвимог буде наведено нижче) та однієї задачі з таким розподілом коефіцієнтів: перша підвимога – 0.5, друга – 0.3, третя – 0.1 та задача – 0.1. Вимога створена 12.05, станом на 14.05: перша підвимога виконана на 30%, друга – без змін, тобто – 0%, третя – 70% та задача – 50%, отже, стан вимоги дорівнює: 30*0.5 + 0*0.3 + 70*0.1 + 50*0.1 = 27%. Станом на 17.05: перша – 70%,

друга – 40%, третя – 100% та задача – 90%, отже, стан вимоги дорівнює: 70*0.5 + 40*0.3 + 100*0.1 + 90*0.1 = 66%.

При автоматичній зміні стан підвимоги розраховується з урахуванням вагових коефіцієнтів задач, що задаються при створенні. Розрахунок виконується аналогічно до розрахунку вимоги.

У випадку ручного проставлення стану підвимоги, менеджер проекту вручну виставляє з деяким інтервалом у часі процент виконання. Також, слід зазначити, що в обох випадках (автоматично та вручну) менеджер може або зобов'язується (якщо вручну) створювати робочі записи до підвимоги, наприклад, 14.03 – «було закінчено створення дизайну додатка». При такій зміні, стан вимоги буде кожен раз перераховуватися автоматично. Аналогічні дії по зміні стану можна здійснювати для вимоги.

Під «Аукціоном вимог» слід розуміти частину функціоналу системи, що відповідає за процес переговорів та затвердження умов (суть, терміни виконання, ціна) вимоги між сторонами замовника та розробників (в контексті використання типових ролей, дана відповідальність належить менеджеру проекту). Головна ідея полягає у тому, що замовник та менеджер введуть перемовини про умови вимоги, щоразу її деталізуючи та уточнюючи. В рекомендованому розвитку подій (згідно з концепцією СПЗ) вимога береться до опрацювання лише коли має двостороннє погодження. Розглянемо на прикладі. Замовник створює вимогу «Розробити систему для інтеграції з хмарними обчисленнями» та виставляє ціну у 10000 у.о. з термінами до 30.12, про це сповіщається менеджер проекту, що не згоден з ціною, він її підвищує до 15000 у.о. та обов'язково додає коментар, наприклад, «Необхідні додатковий розробник». Далі про це споващується замовник, вирішує, що вирішує, на додаткового розробника необхідно замість 5000 у.о., 2000 у.о. Знижує ціну до 12000 у.о. і продовжує терміни виконання до 07.01 та дає зі своєї сторони згоду. Менеджер бачить зміни в ціні та термінах, приймає рішення, що це його влаштовую та дає зі своєї сторони згоду. Після цього вимога буде затверджена і перейде у

стадію виконання, що супроводжується створенням задач (нижче про це буде описано).

Життєвий цикл вимоги складається з наступних етапів: створена («To Discuss») – обговорюється («In Discuss») – обговорення завершено («Accepted») – почалася розробка («In Progress») – завершено виконання вимоги («Completed»). Також передбачається два додаткових етапи перехід до яких можливий із етапів In Discussing, Accepted, In Progress: обговорення чи виконання відкладено до невизначених термінів («Frozen») та обговорення чи виконання відмінено («Canceled»).

Для роботи з вимогами та підвимогами передбачено дві додаткові дії такі як розгалуження («Branching») та злиття («Merging»). Під розгалуженням слід розуміти процес декомпозиції вимоги чи підвимоги на дві або більше тих чи інших. Під злиттям слід розуміти зворотній процес до розгалуження – об’єднання двох або більше вимог чи підвимог. Концепція СПЗ рекомендує виконувати дані дії до етапу Accepted, оскільки виконання даних операцій після даного етапу потребує додаткових обговорень та можливо витрат і зміщень кінцевих терміни.

До основних можливостей підсистеми обліку задач відносяться такі функції як: створення, редагування, видалення задач та підзадач, сповіщення про стан виконання задачі або підзадачі (під станом задачі або підзадачі слід розуміти процент закінченості тієї або іншої).

Можливості, що надаються для роботи з задачами та підзадачами згідно з обраною роллю (типовою) описані у розділі 2.2. Під задачею розуміється певний набір завдань, що задовольняють вимогам чи підвимогам (одній чи більше), або ж, навпаки, декілька задач на одну вимогу чи підвимогу. Задача створюється менеджером проекта та ставиться керівнику команди, той в свою чергу, розбиває її на підзадачі та назначає розробників. Під підзадачею розуміється деякий набір робіт (одна чи більше), що безпосередньо виконує розробник.

На задачі та підзадачі є можливість виставляти терміни виконання. Під час виконання підзадачі розробник сповіщує про її стан виконання. Наприклад,

підзадача була створена та взята до виконання 07.10.2017 року, потрібно її виконати до 17.10.2017 року. Під час розробки, розробник сповіщує про процент виконання: 08.10 виконано 7%, 10.10 – 15%, 11.10 – 37%, 15.10 – 83%, 17.10 – 100%. Також слід зазначити, під час звітування, розробник має можливості створювати робочі записи, наприклад:

— 08.10 – «було проаналізовано, які треба внести зміни до архітектури»;

— 11.10 – «було створено основний каркас в архітектурі для подальшої розробки»;

— 15.10 – «було реалізовано функції створення та редагування профілю користувача»;

— 17.10 – «було реалізовано видалення профілю користувача, завершено виконання підзадачі».

Після того як задача була розбита на підзадачі та взята до виконання розробники сповіщують про стан виконання підзадача (як було вище описано). Стан виконання задачі може автоматично (з урахуванням вагових коефіцієнтів чи часовим розподіленням, також може застосовуватися для підзадачі), або ж виставлятися вручну керівником команди («типова» роль, див. розділ 2.2).

Автоматична зміна стану виконання задачі з урахуванням вагових коефіцієнтів здійснюється аналогічно до зміни стану вимогу (див. розділ 2.3.1). Автоматична зміна стану виконання задачі чи підзадачі з попередньо визначеним часовим розподілом передбачає можливість налаштувати автозміну стану погодинно чи подобово, тобто, запланувати в які дні на скільки повинен зрости стан чи використати лінійний або експоненціальний розподіл зростання стану.

У випадку ручного проставлення стану задачі, керівник команди вручну виставляє з деяким інтервалом у часі процент виконання. Також, слід зазначити, що в обох випадках (автоматично та вручну) керівник може створювати робочі записи до задачі (аналогічно як і з підзачами).

При внесенні змін до стану підзадачі, керівник проекту сповіщується про них, а про зміни у задачі – менеджер проекту (у випадку користуванням вбудованими

ролями). Фіксація змін та їх сповіщення здійснюється за допомогою сервісу сповіщень – про це написано нижче.

Життєвий цикл задачі та підзадачі складається з наступних етапів: створена («To Do») – прийнята до виконання («In Progress») – виконання закінчено («Completed») – тестування пройдено («Accepted»).

До основних можливостей підсистеми обліку дефектів відносяться такі функції як: створення, редагування, видалення дефектів. Під дефектом слід розуміти деякий недолік системи, що знайдено під час проведення тестування. Дефект може бути створений під час тестування задачі або підзадачі, що знаходяться у етапі «Completed» або ж під час проведення планових тестів. При виявленні недоліку, тестувальник створює дефект, що закріпляє за розробником, що займався тією підзадачею, що призвела до цього. Також дефект може бути створений під час проведення планових тестувань, наприклад, стрес-тестувань («Stress testing»), системних тестів («System testing») та інші.

До основних можливостей підсистеми динамічних ролей відносяться такі функції як створення, редагування, видалення ролі та її назначення користувачу.

Під динамічної роллю слід розуміти, деяку сутність, що визначає права користувача на такі функціїї як перегляд, створення, редагування та видалення іншої сутності. Згідно з наданами правами створеної ролі, динамічно будується АРМ для користувача. Побудований АРМ відподівідає усім правам ролі до якої належить користувач у рамках одного проекту. Якщо користувач належить до різних проектів з різними ролями, під час виконання система перебудує АРМ відповідно до його прав. Як описувалося у розділі 2.2 система вже містить вбудовані ролі та відповідні їм АРМ.

До основних можливостей підсистеми сповіщень та нагадувань відносяться такі функції як сповіщення користувача у змінах деякої сутності (наприклад, зміна стану виконання задачі), нагадування про заплановані дії та інше.

Для доставки сповіщень використовується сервіс сповіщень (див. розділ 2.1.3). Після автентифікації користувача, він автоматично підписується на отримання

сповіщень, що відповідають проекту(-ів) в якому(-их) він приймає участь та які не суперечять його правам, наприклад, якщо користувач має роль «Розробник», то він не може отримувати сповіщення про внесені зміни у вимогу.

Під нагадуванням слід розуміти сповіщення, що надсилається у випадку коли користувач створив деяку заплановану дію та налаштував час нагадування. Наприклад, менеджер проекту створює 07.12 запис «Мітинг о 17:00 10.12» та налаштовує, щоб нагадування приходило починаючи з 19:00 09.12 по 16:00 10.12 кожну годину окрім «годин відпочинку» (наприклад, з 20:00 по 10:00 сповіщення не будуть приходити). Відповідно менеджер буде отримувати нагадування 19:00 09.12, 10:00 10.12, 11:00 10.12 і так далі до 16:00 10.12.

Для забезпечення комунікації між користувачами в рамках одного проекту передбачено підсистему спілкування. Дана підсистема забезпечує обмін текстовим повідомленням з можливістю прикріпленням додатків між користувачами.

Для того щоб почати спілкування необхідно мати членство у групі. Під групою слід розуміти сутність, в якій знаходяться деякі користувачі. Користувач може знаходитися одночасно в декількох групах. Відповідно він буде отримувати лише ті повідомлення, що відноситься до тієї чи іншої групи в якій він знаходиться.

Для того, щоб додавати користувачів до проекту передбачена можливість генерації запрошення. Наприклад, користувач створив проект та хоче додати керівника команди та розробника. Для цього він створює запрошення, де вказує до якого проекту додати та з якою роллю. Після цього до того кого запрошують надійде сповіщення про нове запрошення у двох формах: в самій системі та на поштову скриньку. Після підтвердження запрошення той, хто запрасовував отримує відповідне сповіщення.

Під час розробки програмного продукту для організації спільного доступу до вихідних кодів додатку та фіксування змін, зазвичай, застосовуються системи версійного контролю.

Система управління версіями – програмне забезпечення для полегшення робіт з інформацією, що часто змінюється. Система версійного контролю надає можливість

зберігати декілька версій одного й того ж самого документа, за необхідністю повертатися до попередніх версій, визначати, хто та коли вніс ту чи іншу зміну.

Станом на 2016 рік згідно з статистикою наданою RhodeCode найпопулярнішою системою управління версіями є Git, далі Mercurial [13]. Найпопулярніші веб-сервіси для Git – GitHub, Mercurial – Bitbucket.

При реєстрації у системі є можливість вказати відповідний обліковий запис, що зареєстрований у веб-сервісі (GitHub чи Bitbucket). Під інтеграцією розуміється можливість автоматично робити відповідність між здійсненими комітами і підзадачею та збереження цієї інформації у базу даних. Також робити підписку на відправку сповіщень про нові коміти у репозиторії від обраних користувачів, за замовчуванням керівник команди отримує сповіщення від розробників, що входять до його команди.

В системі передбачено можливість для перегляду виконаної роботи, побудови статистики та інші засоби для оцінки роботи, що допомогають визначити необхідну суму оплати.

Для зручності оплати вимог та розподілу грошових мас між виконавцями передбачено інтеграцію з зовнішніми сервісами оплати. Передбачається підтримка таких популярних сервісів як Authorize.Net, PayPal, SecurePay.com [14]. Щоб здійснювати оплату, при реєстрації або редагуванні облікового запису необхідно вказати додаткові дані відповідно з обраним сервісом оплати. Щоб отримувати виплати за виконану роботу необхідно вказати відповідні реквізити кінцевого рахунку.

Однієї із ключових цілей системи є відмова від паперових документів, наприклад, технічного завдання. Для досягнення даної цілі необхідно, щоб наша система мала юридичну силу, а наші дані, що зберігаються – вважалися достовірними, щоб ними можна було оперувати під час судових розглядин. Передбачається можливість затвердження інформації цифровим підписом.

Для реалізації даної можливості необхідно залучення фахівця з відповідної галузі та подальша консультація.

2.4 Безпека

Сучасні тенденції в області інформаційної безпеки характеризується стбільним зростанням кількості комп'ютерних атак, що призводять до зниження рівня захищеності ресурсів автоматизованих систем. У більшості випадків основною причиною успішності комп'ютерної атаки є наявність вразливостей програмного забезпечення (ПЗ), що використовуються в складі таких систем. Одним з на правлінь підвищення рівня безпеки є впровадження в рамках життєвого циклу ПО різних процедур, що стосуються зниження числа помилок і вразливостей. Все це визначає актуальну задачу формування вимог до процесів створення, виконання яких дозволить скоротити число вразливостей та забезпечити їх оперативне усунення в разі виявлення. Недоліком програми називається будь яка помилка, допущена під час проектування або реалізації програми, яка в разі залишення її невиправленої, може бути причиною вразливості ПЗ. Під вразливістю програми будемо розуміти недолік програми, який може бути використаний для реалізації загроз безпеці інформації. Слід зазначити, що вразливість програми може бути результатом її реалізації без урахування вимог щодо забезпечення безпеки інформації або результатом наявності помилок проектування або кодування. Треба розуміти, що для реалізації уразливості програми необхідна наявність суб'єкта впливу на програму (порушника) і здатного експлуатувати вразливість. Безпечним програмним забезпеченням будемо називати ПО, розроблене з використанням сукупності заходів, спрямованих на запобігання появи та усунення вразливостей програми.

Розглянемо дві сторони безпеки даних: передача та доступ. Для підвищення захищеності інформації під час передачі використовується протокол HTTPS, що шифрує інформацію передавану у канал.

HTTPS (HyperText Transfer Protocol Secure) – доповнення прокола HTTP, для підтримки шифрування з ціллю підвищення безпеки. Інформація в протоколі HTTPS передається поверх криптографічних протоколів SSL або TLS.

При використанні корпоративної версії дані передаються лише в корпоративній мережі до якої мають доступ обмежене коло користувачів. Завдяки цьому трафік ізолюється і не передається через Інтернет.

Для автентифікації користувачів використовуються токени, що шифруються з використанням симетричних алгоритмів шифрування. Також слід відмітити завдяки використанням ролей (див. розділ. 2.2) можна розмежувати права на доступ до певної інформації. Наприклад, якщо користувач має типову роль «Розробник», за замовчуванням, він не має права переглядати виставлені вимоги.

2.5 Масштабованість

Масштабованість програмного забезпечення - здатність програмного забезпечення коректно працювати на малих і на великих системах з продуктивністю, яка збільшується пропорційно обчислювальної потужності системи (швидкість виконання програм прямо пропорційна продуктивності та кількості процесорів).

Масштабованість програмного забезпечення зачіпає всі його рівні від простих механізмів передачі повідомлень до роботи з такими складними об'єктами як монітори транзакцій і вся середу прикладної системи. Зокрема, програмне забезпечення повинне мінімізувати трафік міжпроцесорного обміну, який може перешкоджати лінійному зростанню продуктивності системи. [18]

Масштабованість програмного забезпечення досягається за рахунок можливості роботи пакетів з різними базами даних для підприємств різного розміру.

Масштабованість означає можливість об'єднання абсолютно будь-якої кількості колись локальних комп'ютерів у мережу, в якій кожен користувач має строго позначені рамки розв'язуваних їм завдань, відповідальності та доступу до інформації.

При зростанні попиту на продукт збільшуються навантаження на систему, щоб уникнути ситуації коли час відповіді перевищує допустимий чи взагалі повного краху використовуються масштабування. Під високими навантаженнями слід розуміти такі навантаженнями з якими не справляється залізо. Наприклад, на нашу систему виріс попит з 500 запитів в секунду до 10000. Щоб справитися з даною проблемою існує два підходи: вертикальне масштабування – купуємо більш потужне «залізо», горизонтальне – купуємо новий «хост», таким чином навантаження буде розподілено між двома серверами [15]. Очевидно, що горизонтальне масштабування перспективніше, оскільки ми завжди можемо докуповувати нові хости, а при вертикальному – ми рано чи пізно все одно досягнемо максимальної конфігурації, а з навантаженням так і не зможемо справлятися.

Якщо в випадку вертикального масштабування нам не потрібно вносити додаткових змін у код, то при горизонтальному – потрібно заздалегідь проектувати та розробляти систему з урахуванням цієї можливості.

Наша система має можливість горизонтально масштабуватися. Наприклад, зі збільшенням запитів, ми з легкістю можемо додати нові «хости» та розподілити навантаження. Використання мікросервісів також дає такі можливості. Наприклад, збільшується кількість сповіщень, що потрібно надсилати, ми можемо просто додати ще декілька копій сервісу сповіщень тим самим розподілити навантаження. Для БД можна використовувати підхід «мультимастер» реплікації. Тобто будь-які дані, що необхідно зберегти будуть записані на два сервери БД.

2.6 Хмарні обчислення

На даний момент існує кілька провайдерів сервісів хмарних обчислень. Для нашого аналізу досить короткого і поверхневого розгляду, візьмемо лише найбільші та представницькі b2b сервіси, які можуть бути використані Інтернет-стартапами у своїй діяльності [6].

Azure Services Platform – знаходиться в розробці сервіс надання віддаленої хмарної платформи, що дозволяє зберігати дані і виконувати веб-додатки на віддаленій хмарі. Над платформою знаходиться так звана «операційна система в хмарі» під назвою Windows Azure, що виробляє керування запуском додатків на безлічі віртуальних машин центру даних Microsoft. Розроблено офіційний набір SDK для Visual studio, що пропонує для розробників досить низькі рівні входу. Офіційно підтримується технологія ASP.NET мови C# і VB.Net ведуться розробки для Java SDK і Ruby.

Amazon Web Services – сервіси виконання високомасштабних додатків, зберігання інформації на віддалених серверах компанії Amazon, надають всі моделі SaaS, IaaS і PaaS.

Google Apps Engine – сервіс компанії Google, поки що знаходиться в стадії публічного бета-тестування, що надає платформу для створення і розгортання додатків на інфраструктурі датацентрів компанії Google. Додаток в хмарі виконується на декількох віртуальних серверах. На даний момент спочатку безкоштовно надаються можливості 5 мільйонів переглядів на місяць, а потім за кожне перевищення стягується відповідна пропорційна плата. Офіційно підтримувані мови: Python і Java. Система так само використовує не реляційну структуру для зберігання баз даних зі своїм SQL-подібних мовою запитів, що має назву GQL.

Salesforce.com – один з найбільших провайдерів, що надає переважно SaaS і PaaS. Надаючи щомісячну передплату, компанія позиціонує себе як провайдер нових видів управління взаємин з клієнтами (Customer Relationship Management). Перекладений на 16 мов, сервіс вже має понад 1,5 млн. передплатників, серед яких Siemens, Dell, Starbucks Coffee та інші. Додатки на платформі можуть бути створені за допомогою спеціального Java-подібної мови Apex, а так само мови Visualforce для роботи з HTML, AJAX і Flex.

В якості провайдера хмарних обчислень розглядається Microsoft Azure [16].

Microsoft Azure добре знайома як відкрита і гнучка платформа хмарних обчислень на рівні компаній і навіть великих корпорацій. Але це ще й постійно зростаюча колекція інтегрованих хмарних служб.

2.7 Вимоги до комплексу програмно-технічного забезпечення

Рекомендованими характеристиками комплексу програмно-технічного забезпечення для різних його компонентів

Для серверу:

— процесор Intel Core i5 з тактовою частотою не менше 2.5 ГГц;
— об’єм оперативної пам’ять від 16 ГБ;
— наявність 1 ГБ вільного місця на жорсткому диску;
— наявність відеокарти;
— операційна система Windows Server 2012R2 та старше;
— встановлений .NET Core 2.1;
— встановлений PostgreSQL починаючи з версії 9.6.

Для шини повідомлень:

— процесор Intel Core i5 з тактовою частотою не менше 2.5 ГГц;
— об’єм оперативної пам’ять від 16 ГБ;
— наявність 4 ГБ вільного місця на жорсткому диску;
— наявність відеокарти;
— операційна система Windows Server 2012R2 та старше;
— встановлений RabbitMQ починаючи з версії 3.6.2.

Для настільного клієнта:

— процесор Intel Core i3 з тактовою частотою не менше 1.8 ГГц;
— об’єм оперативної пам’ять від 4 ГБ;
— наявність 256 МБ вільного місця на жорсткому диску;
— наявність відеокарти;
— операційна система Windows 7 та старше;

— встановлений .NET Framework 4.6.2.

Для мобільного клієнта:

— б’єм оперативної пам’яті від 2 ГБ;

— наявність 128 МБ фізичного вільного місця;

— операційна система iOS 10 чи Android 7 та старше;

Для веб клієнту:

— процесор Intel Core i3 з тактовою частотою не менше 1.8 ГГц;

— об’єм оперативної пам’яті від 2 ГБ;

— наявність відеокарти;

— наявність браузера з підтримкою WebSocket та ECMAScript 6.

3 РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Вибір інструментів розробки

В сучасному світі, коли кількість мов програмування сягнула сотень і навіть тисяч, дуже не просто зробити вибір який буде вірним до кінця проекту. Вимоги можуть змінитися будь якої миті, а змінити інструменти які використовуються зазвичай майже не можливо. Цей фактор і був передумовою появи мікросервісної архітектури. Адаптери між ними могли реалізовуватися просто і швидко, використовуючи майже будь який з розповсюджених протоколів, як то HTTP чи AMQP. Таким чином можна зробити висновок що при правильному проектуванні системи, головними чинниками вибору будуть передумови появу проекту — чи є в штаті компанії розробники і які мови програмування вони використовують, на який термін планується розтягнути підтримку проекту, яку кількість коштів замовник готовий викласти на етапі розробки і т.д.

Для того аби зробити раціональний вибір, варто визначити вимоги до кінцевого продукту що стосуються його технічної складової. Для реалізації необхідних компонентів висунуто наступні вимоги:

– багатоплатформність — на початкових етапах найвигідніше розгортати систему на значно дешевших серверах, наприклад Ubuntu Server, а пізніше за необхідності переглянути це рішення;

– розповсюджена МП з низьким порогом входу — аби розробку можна було підтримувати тривалий час з розумними витратами ресурсів;

– підтримка як спільноти так і корпорацій — кількість доступних рішень для такої мови буде достатньо велика.

Зважаючи на перелічене та на досвід автора було обрано мову програмування C#, а за основну технологію для написання компонентів ASP.NET Core, через

сумісність з .Net Standard що значно збільшує кількість доступних публічних розширень та рішень.

Для роботи з базою даних було обрано бібліотеку типів Enitity Framework Core яка надає зручний доступ до даних БД та має велику швидкодію. Для розробки клієнтського застосунку для АРМ було обрано технологію розробки класичних застосунків Windows Presentation Foundation(WPF) оскільки надає змогу використання останніх технологічних рішень для UI. На жаль, в межах цієї дисертації не було змоги розробити застосунок для операційних систем відмінних від Windows, проте саме ця ОС є найпопулярнішою для АРМ-ів.

Як середовище розробки, після зробленого раніше вибору, було обрано Visual Studio Community яка надає абсолютно всі необхідні інструменти розробки, а також є багатоплатформним.

3.2 Проектування структури БД

Ми обрали базу даних PostgreSQL, оскільки вона кросплатформена та має вбудовані інструменти для горизонтального масштабування такі як шардинг та «мультимастер». При проектуванні таблиць використовувалася третя нормальна форма.

Нормальна форм – вимога, яка пред’явлена до структури таблиць в теорії реляційних баз даних для усунення з БД надлишкових функціональних залежностей між атрибутами (полями таблиць). Ціль нормалізації полягає у тому, щоб виключити дублювання даних, яке можне стати причиною отримання неочікуваних результатів при редагування, додаванні чи видаленні. Часто застосовують перші три нормальні форми (НФ).

Відношення знаходиться у 1НФ, якщо його атрибути є простими та атомарними.

Відношення знаходиться у 2НФ, якщо воно знаходиться в 1НФ та кожен не ключовий атрибут залежить від первинного ключа (ПК).

Відношення знаходиться у 3НФ, якщо воно знаходиться в 2НФ та кожен не ключовий атрибут нетранзитивно залежить від ПК.

Розглянемо деякі ключові відношення. Ми маємо користувача (таблиця «User»), який може приймати участь у декількох проектах (таблиця «Project») з різними ролями (таблиця «Role»). Відповідно для розв'язки між ними створено таблицю «UserRoleProject» (рисунок 3.1), яка містить три зовнішні ключі – UserId, ProjectId, RoleId.

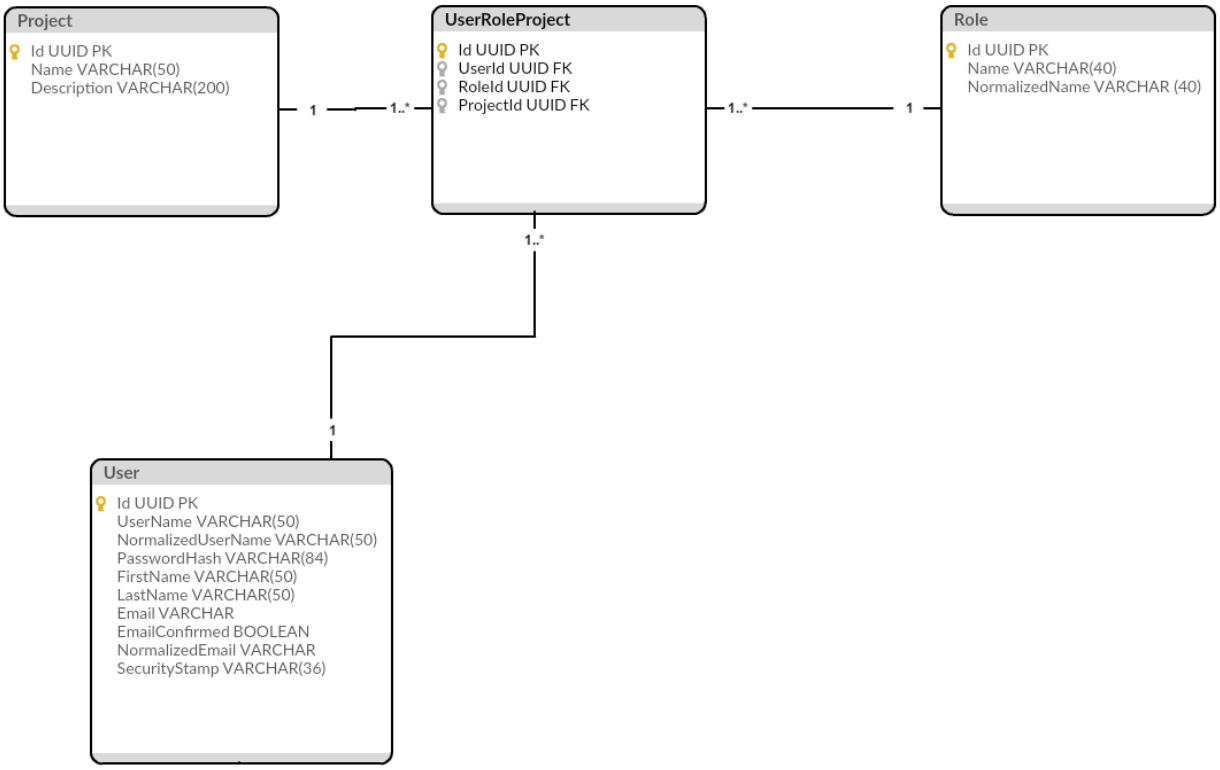

Рисунок 3.1 – Відношення між таблицями «User», «Project», «Role»

Ми надаємо можливість запрошувати інших користувачів до проекту для цього створено таблицю «Invite» (Рисунок 3.2). Вона має два зовнішні ключі на таблиці «User» - це пов'язано з тим, що нам необхідно знати хто створив запрошення (CreatorId) та кого запрошують (InviteeId).

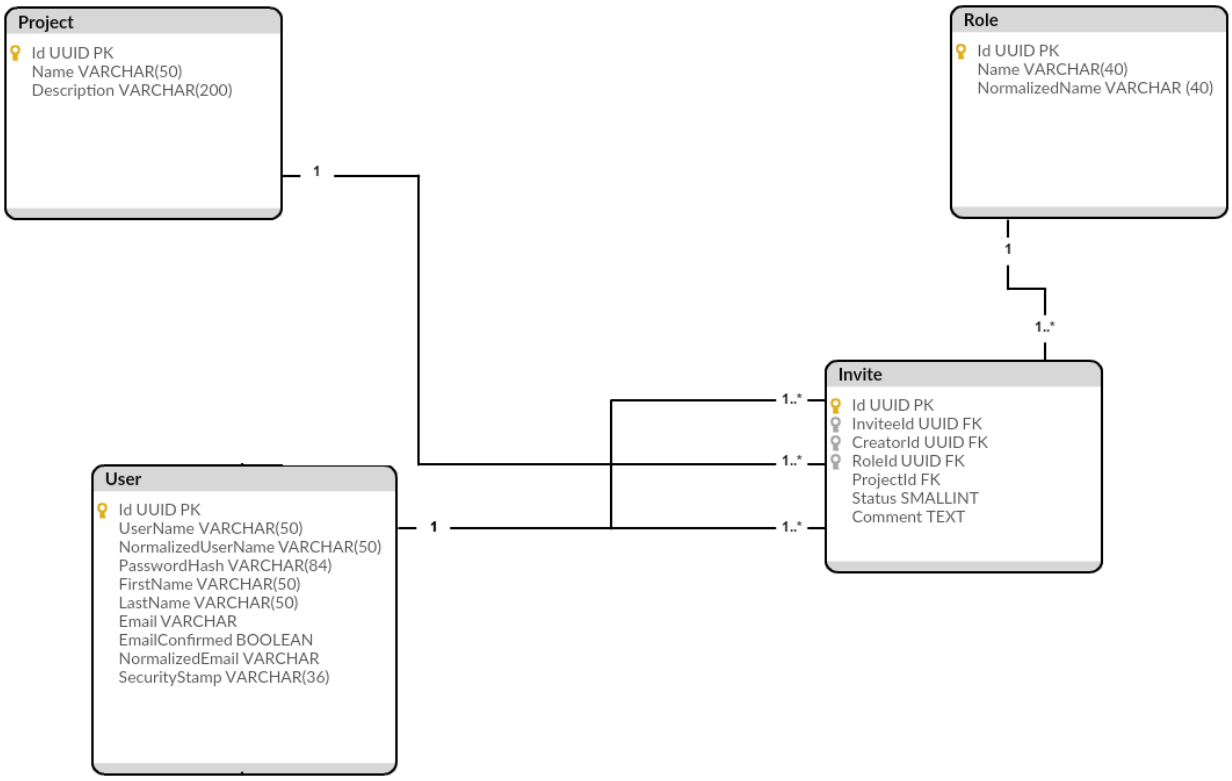

Рисунок 3.2 – Відношення таблиць «User», «Invite», «Role», «Project»

Для збереження бінарних даних створено таблицю «Attachment». Файли можуть бути прикріплені як до вимог так й до задач і для економії місця та скорочення часу очікування надається можливість обрати вже з завантажених файлів, отже, маємо відповідні розв’язочні таблиці зв’язку «many-to-many» «RequirementAttachment» та «TaskAttachment».

3.3 Реалізація серверної частини

Для серверної архітектури було логічним використати інструменти надані обраними технологіями «з коробки» такими як Model-view-controller. ASP.NET Core включає в себе фреймворк MVC, який об'єднує функціональність MVC, Web API і Web Pages. У попередніх версії платформи дані технології реалізувалися окремо і тому містили багато дублюючої функціональності. Зараз же вони об'єднані в одну програмну модель ASP.NET Core MVC. А Web Forms повністю пішли в минуле.

Крім об'єднання вищезазначених технологій в одну модель в MVC був доданий ряд додаткових функцій. Однією з таких функцій є тег-хелпери (tag helper), які дозволяють більш органічно поєднувати синтаксис html з кодом C#. MVC зазвичай використовується для розробки програмного забезпечення, яке розділяє програму на три взаємопов'язаних частини. Це робиться для відокремлення внутрішніх представлень інформації від способів подання та отримання інформації від користувача. Шаблон дизайну MVC відокремлює ці основні компоненти, що дозволяє ефективно використовувати код та вести паралельну розробку.

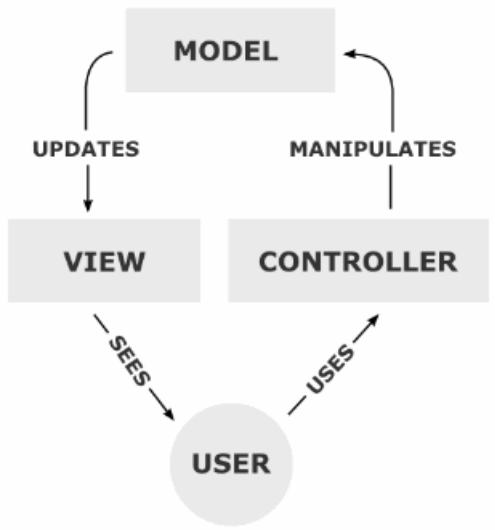

Рисунок 3.3 – Взаємодія користувача, контролера та моделі відображення [31]

Якщо підсумувати, то можна виділити наступні ключові відмінності ASP.NET Core від попередніх версій ASP.NET:

– новий легкий і модульний конвеєр HTTP-запитів;

– можливість розгортати додаток як на IIS, так і в рамках свого власного процесу;

– використання платформи .NET Core і її функціональності;

– поширення пакетів платформи через NuGet;

– єдиний стек веб-розробки, що поєднує Web UI і Web API;

– конфігурація для спрощеного використання в хмарі;

– вбудована підтримка для впровадження залежностей;

– багатоплатформеність — можливість розробки і розгортання додатків ASP.NET на Windows, Mac і Linux;

– розвиток як open source, відкритість до змін.

Внутрішнє представлення даних є одним з ключових факторі для майбутньої розробки та підтримки проекту, оскільки помилка допущена на етапі проектування може призвести до складнощів або навіть унеможливлення реалізації нових вимог.

3.4 Реалізація клієнтської частини

На даний момент реалізовано клієнтський додаток під ОС Windows. Клієнтська частина як і серверна виконана за використанням трьохшарової архітектури (див. розділ 2.1.2). Розберемо реалізацію кожного шару.

Шар доступу до даних являє собою реалізацію REST-клієнта, що отримує та надсилає дані до серверу. Маємо три основні сутності такі як RestClient, CleanSlateRestClient та сервіси, що являють собою реалізацію взаємодії з контролерами.

RestClient – реалізує основні HTTP методи такі як GET, POST, PUT, DELETE і має різні варіації використання. Для реалізації даного клієнту використовувалася бібліотека RestSharp, що надає можливість створення HTTP запитів.

CleanSlateRestClient – агрегує в собі усі сервіси та створює їх реалізації. Якщо порівняти його з реалізацією шару доступу до даних на серверній частині, то він дуже схожий на шаблон UnitOfWork.

Сервіси слугують для виклику відповідного методу в контролері на сервері та передають на шар бізнес-логіки готові об’єкти. Наприклад, маємо контролер AccountController, що має метод Token який необхідно викликати з використання HTTP методу POST. На клієнтській частині маємо AccountService, що має метод GetToken. Задача методу GetToken отримати з верхнього шару ім’я користувача («username») та пароль, викликати метод POST у RestClient, потім отримати дані та

передати назад до шару бізнес-логіки. Кожен клас сервісу, який призначен для комунікації з верхнім шаром має закінчення «Service».

Шар бізнес-логіки для первинної перевірки відповідності отриманих даних від шару презентації з відповідністю до бізнес-правил та виклику необхідного методу з шара доступу до даних й віддача отриманих даних до верхнього шару.

Первинна перевірка відповідності отриманих даних з відповідністю для бізнес-правил зроблена для зменшення кількості запитів до сервера й скорочення часу відповіді. Якщо надійшли невірні данні користувач негайно отримає результат тому, що не буде створено запит до сервера. Кожен клас, який призначен для комунікації з шаром презентації має закінчення «Manager».

Шар презентації виконано як клієнтський додаток під ОС Windows з використанням технології WPF. В основі реалізації закладено шаблон MVVM (рисунок 3.4). Він складається з трьох основних частин:

– Модель (Model) – являє собою бізнес логіку та фундаментальні дані, що необхідні для роботи додатку;

– Представлення (View) – це графічний інтерфейс, тобто, вікно, кнопки тощо. Представлення робить підпис на події зміни значень властивостей чи команд, які надає Модель Представлення. Якщо в Моделі Представлення змінилось будь-яка властивість, то будуть сповіщенні усі хто підписався, та Представлення, в свою чергу, робить запит на оновлено значення властивості з Моделі Представлення;

– Модель Представлення (ViewModel) – являє собою з однієї сторони, абстракцію Представлення, а з іншої, надає обгортку даних із Моделі, що необхідно прив’язати. Тобто, вона містить Модель, що конвертована до Представлення, також містить в собі команди, якими може користуватися Представлення, щоб змінити Модель.

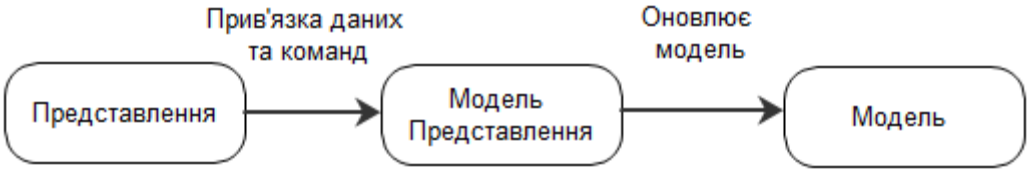

Рисунок 3.4 – Шаблон MVVM

В нашій реалізації Модель – це шар бізнес логіки, Представлення – це XAML розмітка вікна чи користувацького компонента (UserControl), Модель Представлення – це зв'язна ланка між ними.

Для підтримки шаблону MVVM та забезпечення прив’язки даних використовувалася бібліотека MVVMLight. Вона має достатній набір інструментів, що необхідні для реалізації шаблона MVVM.

Для забезпечення комунікації між ViewModel використовувалися повідомлення. Тобто, з однієї сторони ми маємо видавця – той хто створює повідомлення, з іншої споживача – той хто отримує повідомлення того типу на який він зробив підписку. Для реалізації даної схеми використовувався Messenger, який надає бібліотека MVVMLight. Він має два головних методи: Register та Send. Перший робить підписку повідомлення з вказанням їх типу, другий відправляє повідомлення (рисунок 3.5).

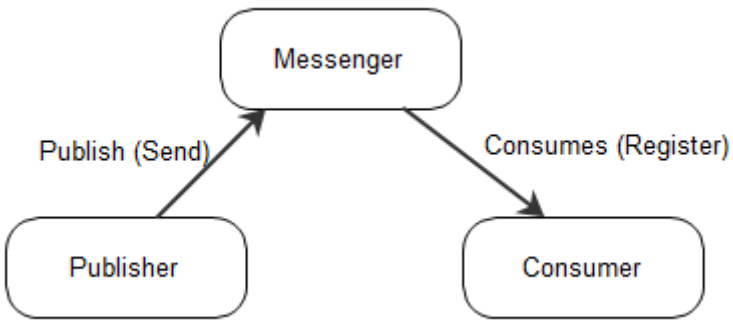

Рисунок 3.5 – Схема роботи Messenger

Для забезпечення зміни графічних форм було розроблено власний навігаційний модуль. Він складається з чотирьох класів (рисунок 3.6):

– NavigationViewResolver – його задача це реєструвати (метод «Register») графічну форму й за необхідності її контекст даних (data context) та відавати її при запиті (метод «Resolve»);

– NavigationController – його задача це змінювати графічні форми у NavigationControl та за необхідності контекст даних;

– NavigationControl – його задача це показувати користувачу графічні форми у відповідності з NavigationController;

– NavigationService – його задача це надати інтерфейс для зміни контексту відображення, зміни графічних форм, можливості повернення до попередньої форми. Вихідний код наведено у додатку В.

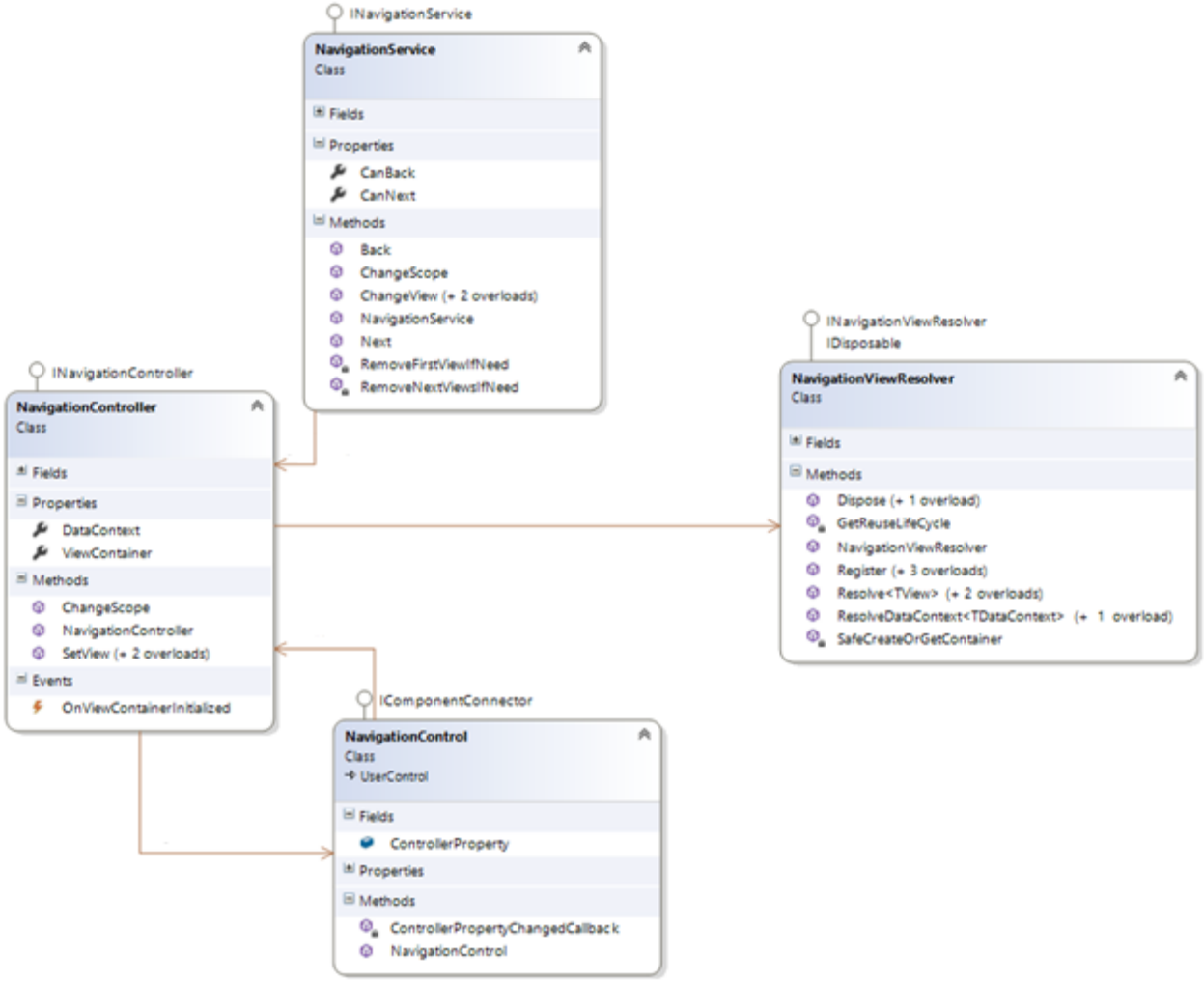

Рисунок 3.6 – Діаграма класів навігаційного модуля

Контекст відображення – це контекст, згідно з яким віддаються ті чи інші екземпляри форм та їх контексту даних. Наприклад, в нашому додатку передбачено, що користувач може приймати участь у декількох проекту, відповідно може мати різні ролі і звичайно ж різна інформація повинна відображатися користувачу. Тобто, в нашому випадку контекстом відображення є ідентифікатор проекту.

Для реєстрації графічної форми та її контексту даних використовувався IoC-контейнер DryIoc. Він також підтримує три стратегії віддачі об’єктів (див. розділ 3.3). Також цей контейнер застосовувався для співставлення абстракцій та їх реалізації, наприклад, IAccountManager та AccountManager.

В основу стилю графічного інтерфейсу закладено «матеріальний дизайн» («material design») розроблений компанією Google. Даний стиль є популярним та використовується у таких відомих веб, мобільних та десктопних додатках як Google Play, Gmail, Youtube, Telegram, ВКонтакте та інших.

3.5 Реалізація сервісу сповіщень

Для реалізації сервісу сповіщень використовувався ASP .NET SignalR. Він надає можливість створити двонаправлений зв’язок між клієнтом та сервісом через WebSocket, що надає можливість сервісу в будь-який момент надсилати сповіщення до клієнта.

Для реалізації комунікації між сервером та сервісами використовувався RabbitMQ. RabbitMQ – це платформа, що реалізує систему обміну повідомленнями між компонентами програмної системи (Message Oriented Middleware) на основі стандарту AMQP [28].

Варто відмітити, що RabbitMQ має вбудовані можливості для горизонтального масштабування, наприклад, створення декількох черг чи під’єднання більшої кількості споживачів на один і той же тип повідомлення.

3.6 Логічна структура модулів

При розробці системи з самого початку був поділ на три основних частини:

- серверна;
- клієнтська;
- спільна.

Серверна частина додатково поділяється на кожний з розроблюваних сервісів та спільні модулі доступу до бази. З архітектурної точки зору, для мікросервісів це є не припустимо, але на даному етапі це можна використати з умовою подальшого розділення баз даних для кожного сервісу.

Всередині модулі поділяються на доменну, доступ до даних та так звані адаптери — взаємодія з зовнішнім світом(по відношеню до кожного з сервісів).

4 ТЕХНОЛОГІЧНИЙ РОЗДІЛ

4.1 Керівництво адміністратора

У даному розглянуто основні функції системи такі як реєстрація, автентифікація, створення та підтвердження запрошень, створення проекту, створення та перегляд вимог.

Для початку роботи з системою, користувач повинен автентифікуватися у системі. На рисунку 4.1 зображено форму автентифікації користувача.

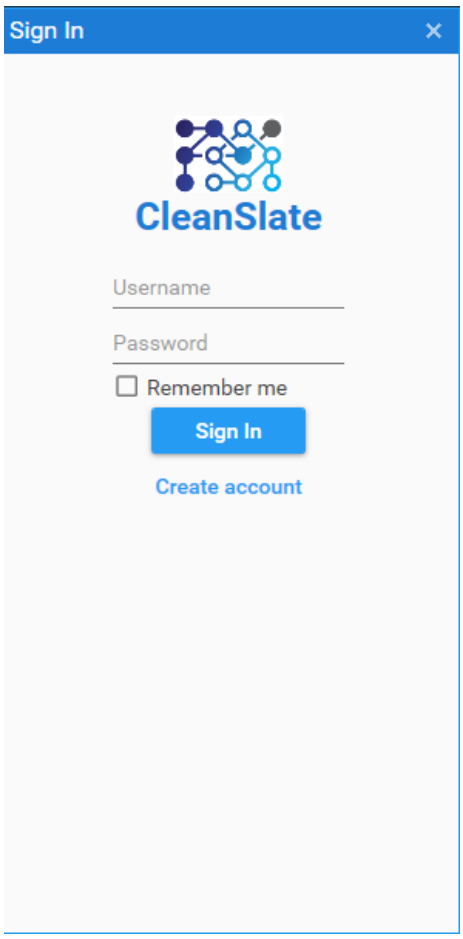

Рисунок 4.1 – Графічна форма автентифікації користувача

Якщо користувач не зареєстрований у системі, він може перейти до форми реєстрації (рисунок 4.2), натиснувши кнопку «Create account», де потрібно ввести необхідні поля для реєстрації.

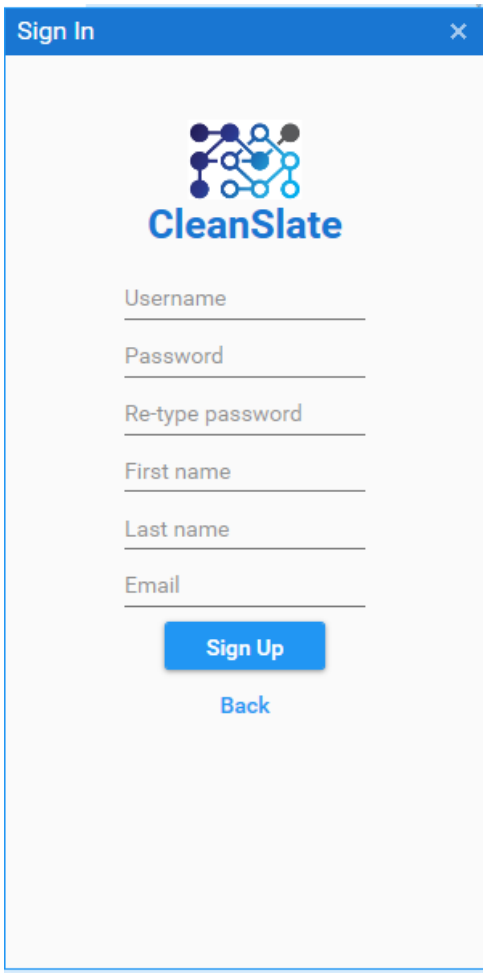

Рисунок 4.2 – Графічна форма реєстрації користувача

Після автентифікації відкривається головна форма (рисунок 4.3).

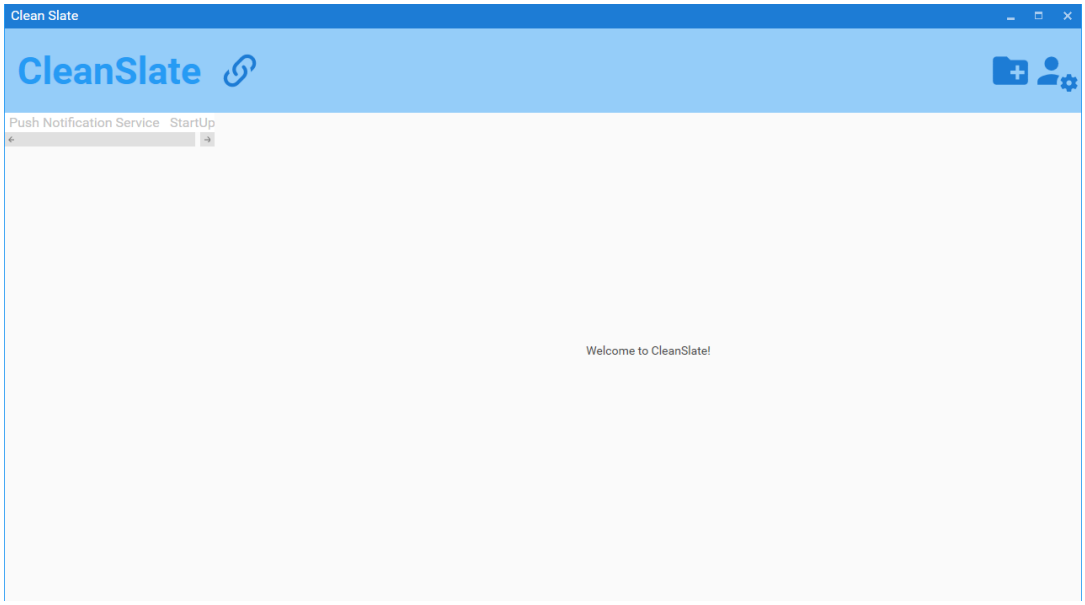

Рисунок 4.3 – Головна форма

Користувач має можливість створити новий проект (рисунок 4.4).

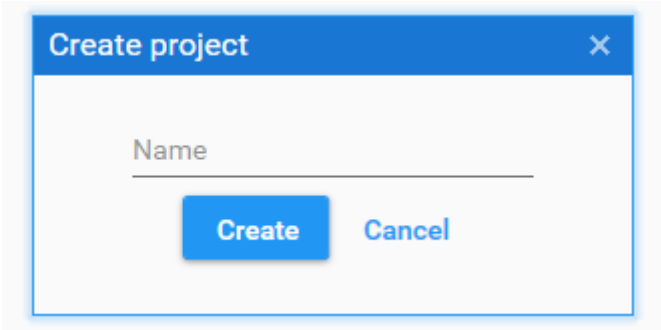

Рисунок 4.4 – Створення нового проекту

На рисунку 4.5 зображено форму для створення запрошення для іншого користувача до проекту.

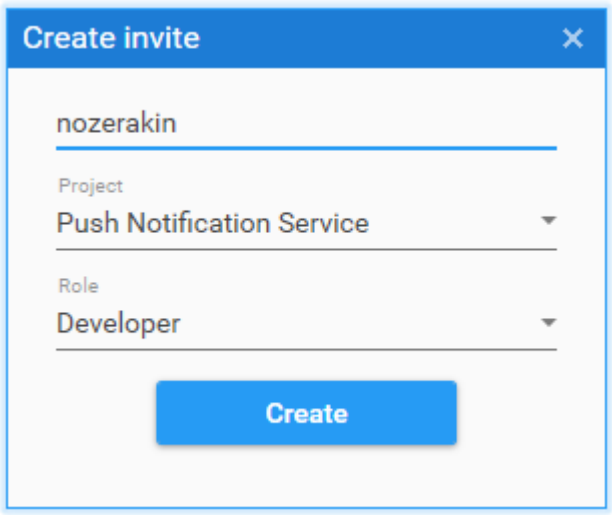

Рисунок 4.5 – Форма створення запрошення

Користувач може переглянути вхідні запрошення та відправлені запрошення на вкладках «Incoming» та «Outgoing», відповідно (рисунок 4.6).

Рисунок 4.6 – Графічна форма з запрошеннями

На рисунку 4.7 зображено графічну форму для створення вимог.

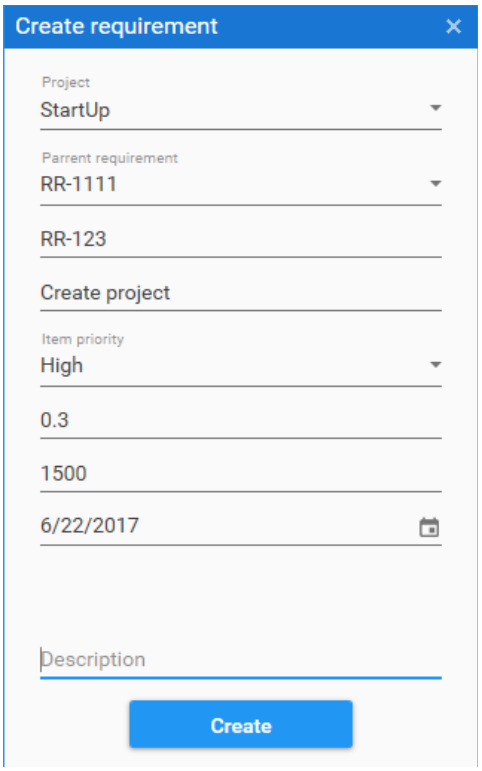

Рисунок 4.7 – Створення нової вимоги

Усі вимоги можна переглянути натиснувши кнопку «Requirements». Вимоги будуються у деревоподібній структурі, натиснувши на елемент у дереві, можна детально переглянути саму вимогу (рисунок 4.8).

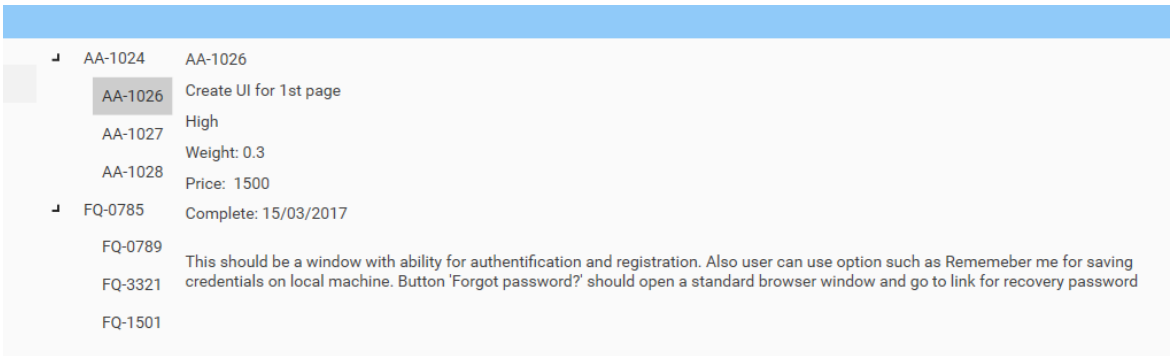

Рисунок 4.8 - Перегляд вимог

4.2 Регресивне тестування

Регресійне тестування програмного забезпечення - це основний спосіб виявлення таких дефектів після закінчення розробки. Регресивні тести виконують як тестувальники, так і розробники. У agile командах, які застосовують методи безперервної інтеграції, автоматизовані регресивні тести запускаються в нічних або тижневих збірках і є відмінним доповненням до юніт-тестів.

Одним з основних факторів успіху даного виду тестування є повнота тестового покриття. Якщо покриття буде недостатнім - тестувальники можуть не знайти критичних дефектів. А якщо покриття буде надлишковим - тестування буде споживати більше часу і грошей, а також виросте час випуску продукту. Тому в підготовці тестового покриття повинні брати участь професійні тестувальники з великим досвідом роботи. Фахівці Перфоманс Лаб проходять навчання і сертифікацію по міжнародній системі ISTQB перш ніж допускаються до створення тестового покриття. Крім цього, Перфоманс Лаб застосують кращі практики регресійного тестування, включаючи інструмент МКС (матрицю критичності / складності), яка дозволяє не тільки підготувати оптимальне покриття, але і пріоритезувати тест-кейси з критичності і часу написання.

Іншим фактором успіху, є залученість предметних фахівців, які добре розуміють суть роботи програмного продукту. Необхідно узгодити з ними тест-кейси перед тим, як команда приступить до їх розробки.

4.3 Модульне тестування

Модульні тести (Unit-тести) надають можливість швидко та автоматично протестувати окремі ділянки коду незалежно від іншої частини програми. При правильному написанні модульних тестів вони в повній мірі можуть покрити більшу частину коду додатку. Більшість юніт-тестів мають наступний ряд ознак: тестування

невеликих ділянок коду («юнітів»), тестування в ізоляції від іншого коду, тестування лише загальнодоступних кінцевих точок, автоматизація тестування. Розглянемо кожну більш детально.

При створенні модульних тестів обираються невеликі частни коду, які необхідно протестувати. Як правило, ділянка коду, яка перевіряється повинна бути менше класу, а у більшості випадків тестується окремий метод класу. Завдяки цьому написання модульних тестів займає небагато часу.

При тестуванні важливо ізолювати перевіряємий код від іншої програми з якою він взаємодіє, щоб потім чітко визначити можливість помилок саме в цій ділянці ізолюваного коду. Це полегшує та підвищує контроль над окремими компонентами програми.

Невеликі зміни в класу можуть призвести до провалу багатьох модульних тестів, оскільки реалізація класу змінилася. Щоб уникнути цього, при написанні юніт-тестів обмежуються тільки загальнодоступними кінцевими точками, що дозволяє ізолювати модульні тести від багатьох деталей внутрішньої реалізації.

Написання модульних тестів для невеликих ділянок коду призводить до того, шо кількість цих юніт-тестів досягає великої кількості. Якщо процес отримання результату та виконання тестів не автоматизован, це може призвести до зниження продуктивності під час розробки. Тому дуже важливо, щоб модульні тести являли собою просте рішення, що надає інформацію чи пройден тест чи ні. Для автоматизації процесу розробники зазвичай використовують готові фреймворки.

Розробка через тестування (Test-Driven Development, TDD) – це підхід, який застосовують при розробці. Його головна ідея полягає в тому, що спочатку пишуть модульні тести, а потім вже програмний код, якого достатнь для виконання цих тестів.

Використання TDD дозволяє знизити кількість потенціних помилок у програмі. Створюючи модульні тести до початку написання коду, ми тим самим описуємо спосіб поведінки майбутніх компонентів, не зв’язуючи себе при цьому з конкретною

реалізацією цих перевіряємих компонентів. Таким чином, тести допомогають оформити та описати API майбутніх компонентів.

Моделі тестів Arrange-Act-Assert являє собою не лише особливість тестування в Visual Studio, а й цілу парадигму тестування:

– arrange – підготовка середовища, в якому виконується код;
– act – тестування коду
– assert – переконуємося, що результат тесту саме той, який ми очікували.

Під час тестування досить часто використовуються фіктивні об’єкти (mock-об’єкти). Mock-об’єкт – це тип об’єктів, які реалізують задані аспекти програмного середовища, яке моделюється. Фіктивний об’єкт являє собою лише конкретну фіктивну реалізацію інтерфейсу, призначену виключно лише для тестування взаємодії та щодо якого висловлюється твердження.

Для написання модульних тестів ми обрали фреймворк NUnit. Для позначення класу, який призначен для юніт-тестів використовують атрибут TestFixtureAttribute. Метод, який являє собою виконання самого тесту декорують атрибутом TestAttribute. Також є можливість позначити метод, задача якого виконати базові налаштування середовище, для цього метод позначають атрибутом SetupAttribute й зазвичай називають Setup. На рисунку 4.8 наведено результат виконання тестів для RequirmentController (додаток В).

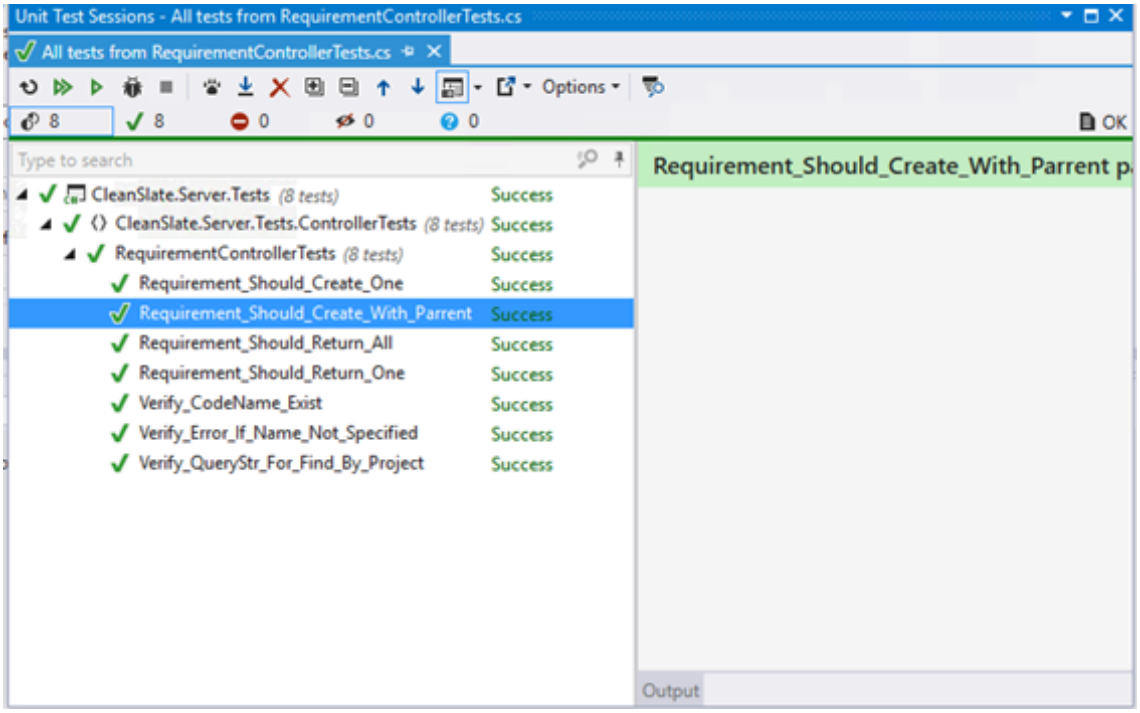

Рисунок 4.8 - Результат виконання модульних тестів

4.4 Інтеграційне тестування

Інтеграційне тестування — це тестування взаємодії декількох класів, виконують разом якусь роботу. Однак як за таким визначенням тестувати не зрозуміло. Можна, звичайно, відштовхуватися від інших видів тестування. Але це загрожує.

Якщо до нього підходити як до unit-тестування, у якого в тестах залежності не замінюються mock-об'єктами, то отримуємо проблеми. Для гарного покриття потрібно написати багато тестів, так як кількість можливих поєднань взаємодіючих компонент - це поліноміальна залежність. Крім того, unit-тести тестують як саме здійснюється взаємодія (див. Тестування методом білого ящика). Через це після рефакторінга, коли якась взаємодія виявилося виділеним в новий клас, тести руйнуються. Потрібно застосовувати менш інвазивний метод.

Підходити ж до інтеграційного тестування як до більш деталізованому системному теж не виходить. В цьому випадку навпаки тестів буде мало для

перевірки всіх використовуваних в програмі взаємодій. Системне тестування занадто високорівневе.

Даний вид тестування є інтеграційним, так як при перевірці викликається код взаємодії декількох класів. Причому важливий тільки результат взаємодії, а не деталі і порядок викликів. Тому на тести не впливає рефакторинг коду. Чи не відбувається надмірного або недостатнього тестування - тестуються лише ті взаємодії, які зустрічаються при обробці реальних даних. Самі тести легко підтримувати, так як специфікація добре читається і її просто змінювати відповідно до нових вимог.

4.5 Перевірка ефективності розробленого рішення

Для підтвердження ефективності розробленої системи було вирішено провести тестову розробку простого програмного продукту, з визначеними вимогами, з використанням різних підходів.

Виконавцями проектів були асистент кафедри АУТС Хмелюк В.С(далі Виконавець 1). та студентка кафедри АУТС Майер І.В.(далі Виконавець 2), замовником в кожному з проектів був автор дисертації. Виконавець 1 використовував розроблену систему для ведення проекту, опису вимог та рішень. Виконавець 2 використовував традиційні методи комунікацій з замовником, постановки задач та здачі виконаних робіт. З кожним виконавцем велися окремі комунікації, вони не перетиналися і не спілкувалися між собою.

Було розроблено мінімалістичне ТЗ в межах необхідного для дослідження(таблиця 4.1). Проектом для розробки було обрано “Мережевий чат”.

Таблиця 4.1 — Вимоги до ПЗ “Мережевий чат”

№	Функція	Вимоги	Термін виконання

1	Чат сервер	запуск сервера на ПК з ОС Windows; прослуховування вхідних підключень; підтримка мінімум 10 з'єднань одночасно;	2 дні
2	Чат клієнт	вибір імені; підключення до серверу; отримання списку чатів;	1 день
3	Обмін повідомленнями	приєднання до обраного чату; надсилання текстових повідомлень; перегляд повідомлень інших користувачів;	1 день
4	Обмін фалами	надсилання файлів до чату;	1 день

Терміни виконання наведені вище з розрахунку 1 день як 8 робочих годин.

Для перевірки ефективності кожного з процесів, після виконання наведених вимог було висунуто додаткову вимогу, про яку виконавцям не було відомо на етапах проектування. Необхідно було внести зміну до основного функціоналу — користувач повинен мати можливість перейти в приватне листування з будь яким іншим користувачем через пряме з'єднання в обхід серверу до якого вони підключені. На виконання цієї вимоги було відведено 2 дні.

За результатами виконання поставлених завдань в повному обсязі було проведено аналіз різних чинників, за головний були виділено го

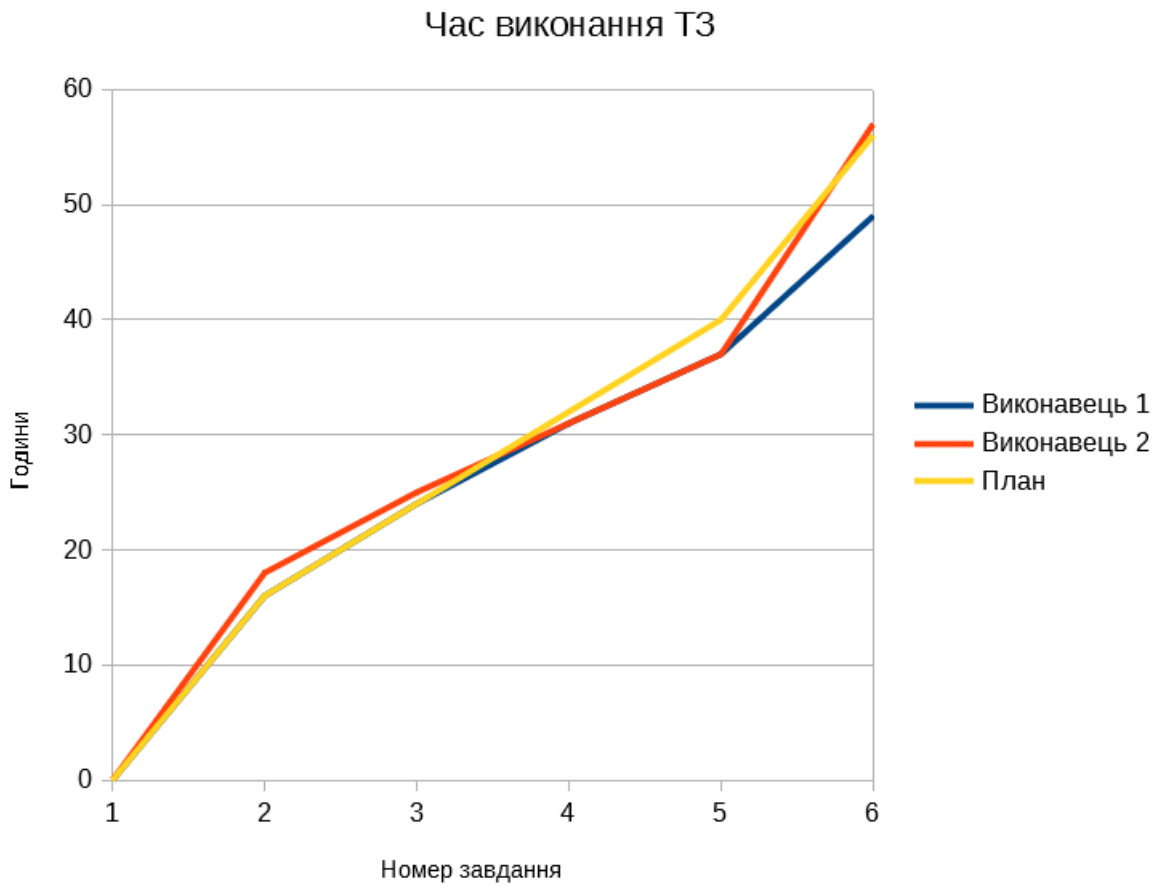

Рисунок 4.8 — Час виконання ТЗ

Як видно з графіку вище, обидва виконавці на етапі виконання донесених ТЗ виконували розробку в межах не великих відхилень від плану, але як тільки було висунуто додаткову вимогу — саме Виконавець 1 зміг швидко узгодити нові вимоги та реалізувати їх навіть швидше запланованого. Виконавець 2 затратив набагато більше часу на узгодження та здачу результатів, що призвело до не великого відхилення від плану.

Таким чином, можна зробити висновок про ефективність розробленої системи, наступним етапом підтвердження має бути повноцінне впровадження системи в існуючий процес та порівняння ефективності старого процесу та нового, з використанням СПЗ.

5 РОЗРОБЛЕННЯ СТАРТАП-ПРОЕКТУ

5.1 Опис ідеї проекту

Таблиця 5.1 – Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Веб-служба для нанесення невидимого цифрового водяного знаку на зображення	1. Захист авторських прав	1. Гарантована ідентифікація автора зображення із відтвореної захисної мітки 2. Захисна мітка не розпізнається без застосування спеціальних засобів 3. Захисна мітка не спотворює оригінальний вигляд зображення 4. Гарантоване відтворення захисної мітки після застосування стандартних перетворень зображення
	2. Знаходження автора зображення	1. Економія часу та ресурсів для пошуку автора роботи 2. Можливість зв’язатися з автором зображення через запропонований веб-портал 3. Можливість ознайомитися з іншими роботами автора
	3. Ідентифікація використання плагіату	1. Економія часу та ресурсів для пошуку неліцензійного використання зображення в інтернеті 2. Підтвердження факту плагіату через відтворення водяного знаку

Продовження таблиці 5.1

	4. Кодування та передача інформації	1. Неможливість розпізнати факт передачі закодованого повідомлення без застосування спеціальних засобів 2. Можливість передати повідомлення по будь-якому каналу передачі даних, що підтримує передачу графічних зображень

Таблиця 5.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів				W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	Digimarc	Visual Watermark	Watermark algid.net			
1.	Вартість експлуатації	54 000 грн/місяць	20,75 млн грн/місяць	60 000 грн/місяць	13 000 грн/місяць	-	-	+
2.	Вартість обслуговування	34 750 грн/місяць	18,55 млн грн/місяць	36 000 грн/місяць	5 000 грн/місяць	-	-	+
3.	Вартість знижки	2 грн/ послугу	0,01 грн/послуг	47,4 грн/ послугу	Відсутня	-	+	-

Продовження таблиці 5.2

4.	Кросбраузерність/ Кросплатформеність	Google Chrome 35.0+, Mozilla Firefox 35.0+, Safari7+, IE10+, Opera11+	Google Chrome 35.0+, Mozilla Firefox 40.0+, Safari7+, IE10+, Opera11+	Windows Vista, 7, 8 10 Mac OS X Lion (10.7)	Google Chrome 27.0+, Mozilla Firefox 34.0+, IE9+, Opera10+	-	+	-
5.	Інтеграція із смартфоном	iOS7.1.2+, Android4.0+	iOS7.1.2+, Android4.0+	Відсутня	Відсутня	-	-	+
6	Робота в мережі	Присутня	Присутня	Відсутня	Присутня	-	+	-
7	Робота в локальній комп’ютерній мережі	Відсутнє	Відсутнє	Присутнє	Відсутнє	-	-	+
8	Формат універсальний вказівник ресурса (URL)	Стандарт W3C	Стандарт W3C	Відсутній	Стандарт W3C	-	+	-
9	Час завантаження сайту	0.5 с	0.3 с	0.5 с	1.2 с	-	+	-
10	Система керування контентом	Стандарт RSS	Стандарт RSS	Стандарт JSR-170	Стандарт RSS	-	+	-
11	Завершенність (вірогідність відмови)	Висока	Середня	Висока	Низька	-	+	-
12	Стійкість до відмов	Середня	Висока	Висока	Низька	-	+	-

Продовження таблиці 5.2

13	Наявність системи резервного копіювання	Присутня	Присутня	Присутня	Відсутня	-	+	-
14	Відновлюваність	Висока	Висока	Висока	Середня	-	+	-
15	Технологічна собівартість	325 000 грн	112,5 млн грн	815 000 грн	75 000	-	+	-
16	Легкість освоєння	Середня	Середня	Середня	Середня	-	+	-
17	Наявність методичних вказівок для використання	Присутні	Присутні	Присутні	Відсутні	-	+	-
18	Автоматична обробка заявки на сервіс	Присутня	Присутня	Присутня	Присутня	-	+	-
19	Наявність служби технічної підтримки	Присутня	Присутня	Присутня	Відсутня	-	+	-
20	Автоматична система оплати сервісу	Присутня	Присутня	Присутня	Відсутня	-	+	-
21	Налаштування параметрів водяного знаку	Присутнє	Відсутнє	Присутнє	Відсутнє	-	+	-
22	Технічна документація	Присутня	Відсутня	Відсутня	Відсутня	+	-	-
23	Надання інтерфейсу для зовнішнього програмного забезпечення	Присутнє	Відсутнє	Відсутнє	Відсутнє	+	-	-
24	Інтеграція з платформами інших сервісів	Присутнє	Частково присутнє	Відсутнє	Відсутнє	+	-	-
25	Генерація унікального ID номера користувача	Присутнє	Відсутнє	Відсутнє	Відсутнє	+	-	-

5.2 Технологічний аудит ідеї проекту

Таблиця 5.3 – Технологічна здійсненність ідеї проекту

№ п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Організація архітектури програмного інтерфейсу додатку	REST (Representational State Transfer)	Наявні	Доступні
2		SOAP (Simple Object Access Protocol)	Наявні	Доступні
3		XML-RPC (Extensible Markup Language Remote Procedure Call)	Наявні	Доступні
Обрана технологія реалізації ідеї проекту: REST (Representational State Transfer)				
4	Реалізація автентифікації користувача	Basic access authentication	Наявні	Доступні
5		OAuth	Наявні	Доступні
6		За допомогою токенів	Наявні	Доступні
7		На основі сертифікатів	Наявні	Доступні
Обрана технологія реалізації ідеї проекту: За допомогою токенів				
8	Реалізація вбудовування водяного знаку в структуру зображення	На основі просторових методів	Наявні	Доступні
9		DCT (Discrete Cosine Transform)	Наявні	Доступні
10		DWT (Discrete Wavelet Transform)	Наявні	Доступні
Обрана технологія реалізації ідеї проекту: DWT (Discrete Wavelet Transform)				

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Таблиця 5.4 – Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	4
2	Загальний обсяг продаж, грн/ум.од	3 грн/ум.од
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (вказати характер обмежень)	Недискримінаційні якісні
5	Специфічні вимоги до стандартизації та сертифікації	Відсутні
6	Середня норма рентабельності в галузі (або по ринку), %	66,5%

Таблиця 5.5 – Характеристика потенційних клієнтів стартап-проекту

№ п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Захист авторського права	1. Виробники графічної продукції 2. Фото сервіси 3. Фото банки	1. Поширення авторських робіт в інтернеті 2. Ведення бізнесу на платформах інтернет-ресурсів	1. Оперативне та гарантоване підтвердження права на власність у разі потреби 2. Збереження первісного вигляду графічної продукції 3. Простий інтерфейс користувача 4. Можливість налаштовувати параметри вбудовування водяної мітки 5. Гнучкі ціни
2	Виявлення порушення авторських прав	1. Виробники графічної продукції 2. Фото сервіси 3. Фото банки	Відсутність контролю поширюваності продукції в інтернет-мережі	1. Пошук інтернет-ресурсів, що використовують неліцензійну продукцію. 2. Підтвердження факту використання плагіату. 3. Простий інтерфейс користувача. 4. Гнучкі ціни.

Продовження таблиці 5.4

3	Пошук автора графічної продукції	1. Споживачі графічної продукції 2. Фото сервіси 3. Фото банки	Прагнення придбати ексклюзивні медіа продукти Різноманітність фото сервісів та фото банків Відсутність можливості ідентифікувати особистість автора через неофіційне джерело	1. Оперативний зв'язок з автором роботи 2. Простий інтерфейс користувача 3. Можливість ознайомитись із іншими роботами автора

Таблиця 5.6 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Крадіжка інтелектуальної власності	Крадіжка ідеї або ключової інтелектуальної інновації	Відсудження прав інтелектуальної власності Забезпечення якіснішого захисту інформації Зміна методики шифрування приватного ключа Попередження користувачів із подальшою співпрацею для мінімізації фактор загрози

Продовження таблиці 5.6

2	Відмова фото сервісів у співпраці	Керівництво фото сервісів не погоджується використовувати сервіси запропонованої веб-служби для захисту авторських прав своїх користувачів	Пропозиція більш вигідних умов співпраці Створення власного фото банку
3	Недостача стартових капіталовкладень	Недостача початкових інвестицій для реалізації мінімально життєздатного продукту	Пошук нових джерел інвестицій
4	Співпраця конкурента із фото сервісами	Фото сервіси використовують веб-служби конкурентів замість запропонованої	Маркетингова кампанія, що висвітлить переваги запропонованого алгоритму вбудовування водяного знаку
5	Недовіра користувачів	Користувачі не довіряють інноваційному способу захисту авторських прав за допомогою проставлення невидимої мітки	Гарантування повернення оплати користувачам у разі не змоги захистити їх авторські права за допомогою запропонованого сервісу

Таблиця 5.7 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Отримання необхідних інвестицій	Сформований початковий капітал, необхідний для реалізації мінімально життєздатного продукту	Розробка мінімально життєздатного продукту
2	Співпраця з відомим фото сервісами	Керівництво фото сервісів із місткістю більше 100 млн графічних матеріалів погодилось використовувати запропоновану веб-службу для захисту авторських прав своїх користувачів	Підтримка стабільної роботи системи та проведення масштабування системи
3	Висока зацікавленість користувачів	Обіг використання веб-служби становить більше 10 000 запитів в день	Підтримка стабільної роботи системи та проведення масштабування системи Збільшення цін на використання сервісу

Продовження таблиці 5.7

4	Успішна маркетингова політика	В результаті проведеної маркетингової політики отримана висока зацікавленість користувачів	Підтримка стабільної роботи системи та проведення масштабування системи Збільшення цін на використання сервісу Використання подібної маркетингової стратегії надалі для залучення нових користувачів
5	Ліквідація конкурента	Конкурент ліквідував свою компанію у результаті власного бажання або зовнішніх чинників	Проведення маркетингової кампанії для монополізації ринку

Таблиця 5.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Олігополія	Незначна кількість конкурентів Велика ринкова сила Схожість використовуваних технологій	Інформування ринку щодо появи нової веб-служби Співпраця із провідними фото сервісами
Галузевий	Загроза появи нових конкурентів Ринкова влада споживачів Висока потреба у товарі	Інформування ринку щодо якості використовуваної новаторської технології Пропозиція гнучких цін

Продовження таблиця 5.8

Внутрішньогалузева	Діяльність в одній галузі економіки Надання сервісів одного типу	Зменшення вартості сервісу Примноження каналів розподілу
Товарно-видова	Надання різних сервісів одного виду	Маркетингова політика
Цінова	Використання цін для покращення економічних умов збуту	Зменшення вартості сервісу Використання нових каналів розподілу
Марочна	Пропозиція схожого сервісу Спільна цільова аудиторія	Інформування ринку щодо якості використовуваної новаторської технології Примноження каналів розподілу

Таблиця 5.9 – Аналіз конкуренції в галузі за М. Портером

	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
Складові аналізу	«Digimarc», «Visual Watermark», «Watermark», «Watermark algid.net»	Розмір капіталовкладень, Забезпечення гнучких цін, Доступ до каналів розподілу, Витрати на масштабах	Відсутні	- Змінні витрати: Виробничі непрямі дегресивні - Системи інформації: пропаганда, реклама та директ-маркетинг, - Рівень чутливості до цін: споживачі орієнтовані на цінність продукту - Продуктова диференціація: якість, спосіб отримання сервісу, швидкість обслуговування Методи контролю якості: тестування та профілювання, прототипування, інспектування коду, аналіз архітектури програмного забезпечення	Копіювання функціоналу, Монополізація дистриб'юторів, Мінімізація цін

Продовження таблиці 5.9

Висновки	CR4 = 95% Індекс Херфіндаля-Хіршмана (HHI) = 6518 Значення показників вказує на високу концентрацію (монополізацію) даного ринку	Можливості входу на ринок забезпечить мінімізація цін, швидкість та простота надавання послуги споживачам і співпраця із головними гравцями ринку. В результаті аналізу проектів на народно-громадських інтернет-платформах потенційних конкурентів знайдено не	Відсутні	Клієнти диктують умови гнучкості цінової політики, високої і довгострокової якості послуг та наявність кооперації із сервісами, що вони використовують	Пропонування вигідних умов дистриб'юторам, забезпечення захисту інтелектуальної власності, гнучкість цінової політики

		було			

Таблиця 5.10 – Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Унікальність сервісу	Для забезпечення захисту авторських прав зображення використовується інноваційна технологія вбудовування невидимої захисної мітки. Наразі не існує сервісів, що використовують дану технологію
2	Модель бізнес для бізнесу	Бізнес модель ґрунтується на співпраці із сервісами, що постачають і керують графічною продукцією та обслуговують запити її власників
3	Цінова політика	Отримання прибутку здійснюється за рахунок отримання процентів з прибутку фото сервісу. Даний підхід дозволить обійти цінову конкуренцію на ринку цільової аудиторії
4	Додаткові послуги	Пошук автора роботи та використання плагіату в інтернеті є унікальним додатковим сервісом на ринку подібних послуг

Таблиця 5.11 – Порівняльний аналіз сильних та слабких сторін «назва проекту»

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з Digimarc						
			–3	–2	–1	0	+1	+2	+3
1	Унікальність сервісу	15						+	
2	Модель бізнес для бізнесу	18							+
3	Цінова політика	10					+		
4	Додаткові послуги	18							+

Таблиця 5.12 – SWOT- аналіз стартап-проекту

Сильні сторони: Якість та довго тривалість послуги Низькі ціни Додаткові сервіси	Слабкі сторони: Недостача стартових капіталовкладень Бізнес модель залежить від політики окремих бізнесів Необхідність стрімкого входу на ринок
Можливості: Інвестиції Реалізація бізнес-моделі Розширений функціонал для фото редакторів Висока зацікавленість цільової аудиторії	Загрози: Крадіжка інтелектуальної власності Відмова дистриб'юторів у співпраці Співпраця конкурента із фото сервісами Недовіра користувачів

Таблиця 5.13 – Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Створення власного фото сервісу	Мало ймовірне	6 місяців
2	Маркетингова кампанія для приваблювання користувачів	Ймовірне	2 місяці
3	Пропонування безкоштовних послуг	Ймовірне	4 місяці
4	Пошук бізнесів іншої галузі для співпраці	Дуже ймовірне	4 місяців
Обрана альтернатива: Пошук бізнесів іншої галузі для співпраці			

5.4 Розроблення ринкової стратегії проекту

Таблиця 5.14 – Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачі в сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Виробники графічної продукції (Фотографи, ілюстратори, художники, типографії, фото-студії)	Висока	83%		Високі бар’єри входу
2	Користувачі графічної продукції (веб-дизайнери, маркетологи, рекламісти, типографії, поліграфічні видавництва)	Середня	65%		Високі бар’єри входу
3	Постачальники та редактори графічної продукції (фото редактори, фото сервіси, фото банки)	Висока	75%		Низьки бар’єри входу
Які цільові групи обрано: Постачальники та редактори графічної продукції, Виробники графічної продукції					

Таблиця 5.15 – Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку*
1	Надання сервісу постачальникам та редакторам графічної продукції у якості веб-служби	Вибірковий розподіл	Здатність протистояти прямим конкурентам Низькі витрати Ефективна співпраця посередників	Стратегія диференціації

Таблиця 5.16 – Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки*
1	Так	Забирати та залучати нових	Розробка плагіну для фото редакторів Інструменти для налаштування параметрів водяного знаку	Стратегія лідера. Розширення первинного попиту

Таблиця 5.17 – Визначення стратегії позиціонування

№ п/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
1	Оперативне та гарантоване підтвердження права на власність у разі потреби Збереження первісного вигляду графічної продукції Простий інтерфейс користувача Можливість налаштовувати параметри вбудовування водяної мітки Гнучкі ціни Пошук інтернет-ресурсів, що використовують неліцензійну продукцію Оперативний зв’язок з автором роботи	Стратегія диференціації	Формування регулярного попиту Збільшення разового використання послуги Виявлення нових груп споживачів Нові напрями застосування існуючої послуги	Захист авторського права Інноваційність технології Простота використання

5.5 Розроблення маркетингової програми стартап-проекту

Таблиця 5.18 – Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити
1	Захист авторського права	Гарантоване виявлення власника роботи Збереження первісного вигляду зображення Низькі ціни	Якість послуги Інноваційність підходу Співпраця з дистриб'юторами Цінова перевага
2	Виявлення порушення авторських прав	Оперативність пошуку плагіату Підтвердження порушення авторських прав	Інноваційність підходу Простота реалізації
3	Пошук автора графічної продукції	Простота зв'язку з автором роботи Можливість ознайомлення з іншими роботами автора	Інноваційність підходу Простота реалізації Глобальність застосування завдяки кількості користувачів

Таблиця 5.19 – Визначення меж встановлення ціни

№ п/п	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
	1.3 – 1.5 грн	6.7- 27.88 грн	7000 грн	1 – 28 грн

Таблиця 5.20 – Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
Закупівля здійснюється через довірені джерела	Інформування користувачів Доступ користування сервісом	Канал одного рівня	Селективна з використанням комбінованого каналу збуту

Таблиця 5.21 – Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
Ведення бізнесу в інтернеті Робота з цифровою продукцією	Прямі неофіційні	Послідовність в реалізації обраної позиції Доступність та об’єктивність інформації про фірму і товар	Формування у цільової аудиторії обізнаності про появу нового сервісу Інформування користувачів про властивості та переваги сервісу Інформування користувачів про нові способи використання відомого сервісу Пояснення цільовій аудиторії принципу дії послуги	Раціоналістична стратегія реклами

ВИСНОВКИ

В магістерській дисертації було спроектовано систему для підтримки життєвого циклу розробки ПЗ як програмного сервісу та реалізовано підсистему, яка забезпечує взаємодію замовників та розробників.

Було розглянуто сім методологій: каскадна, інкрементальна, Agile, RAD, спіральна, RUP, XP. Було наведено їх переваги та недоліки. Порівняльний аналіз допоміг розробити власну концепцію розробки ПЗ – концепція СПЗ.

Було спроектовано систему, яка підтримує життєвий цикл в межах СПЗ. Система пропонує дві версії: публічну та корпоративну. Завдяки використанню трьохшарової та мікросервісної архітектури, вона досить легко горизонтально масштабується. Було запропоновано та розглянуто переваги використання хмарних обчислень, зокрема, Microsoft Azure. Також у подальшому система передбачає можливість надбання юридичної сили, яка допоможе відмовитись від зайвих паперів при обговоренні термінів, умов виконання та оплати робіт.

Було реалізовано підсистему, яка забезпечує взаємодію замовників та розробників при створенні ПЗ. Для реалізації використовувалися сучасні технології програмування.

Було реалізовано кросплатформений сервер. Кросплатформеність надає широкий вибір вибору ОС для розгортання серверу. Він розроблявся з використанням найновітнішого фреймворку від Microsoft – ASP .NET Core. Дані було вирішено зберігати у PostgreSQL, оскільки, він має інструменти для горизонтального масштабування й кросплатформений. Для забезпечення комунікації між мікросервісами було обрано RabbitMQ. Реалізовано сервіс сповіщень, який в режимі реального часу доставляє користувачу сповіщення про події, які відбулися. Було реалізовано настільний клієнт під ОС Windows з використанням WPF.

В цілому, було розроблено підсистему, яка допомагає взаємодіяти замовникам та розробникам. Дана підсистема розроблялася для концепції СПЗ, але вона також підтримує існуючі підходи до розробки ПЗ.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Козловский П., Разработка веб-приложений с использованием AngularJS // Козловский П., Дарвин П. - ДМК Пресс, 2014 – 394 с.

2. Компоненты сетевого приложения. Клиент-серверное взаимодействие и роли серверов. [Електронний ресурс] – Режим доступу до ресурсу: http://www.4stud.info/networking/lecture5.html.

3. Марк Л. Agile Project Management For Dummies / Лейтон Марк. – Hoboken, NJ: For Dummies, 2012. – 360 с. – (1-ше). – (978-1118026243).

4. Офіційний сайт AngularJS. – Режим доступу https://angularjs.org/

5. Паттерн Unit of Work [Електронний ресурс] – Режим доступу до ресурсу: https://metanit.com/sharp/mvc5/23.3.php.

6. Петерсон К. The Waterfall Model in Large-Scale Development / Кай Петерсон. – 386 с.

7. Шор Д. The Art of Agile Development / Джеймс Шор., 2007. – 440 с.

8. Arnold, J. OpenStack swift: Using, administering, and developing for swift object storage. / Arnold, J. // O‘Reilly Media, 2014 – pp. 254 – 256.

9. ASP.NET SignalR [Електронний ресурс] – Режим доступу до ресурсу: https:// www.asp.net/signalr.

10. Azure Notification Hubs [Електронний ресурс] – Режим доступу до ресурсу: https://azure.microsoft.com/en-us/services/notification-hubs/.

11. Azure Service Bus [Електронний ресурс] – Режим доступу до ресурсу: https:// azure.microsoft.com/en-us/services/service-bus/.

12. Beck K. Extreme Programming Explained: Embrace Change / K. Beck, C. Andres.. – 224 с.

13. Entity Framework Core [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/ef/core/.

14. Fowler M. Inversion of Control Containers and the Dependency Injection pattern [Електронний ресурс] / Martin Fowler – Режим доступу до ресурсу: https://martinfowler.com/articles/injection.html.

15. Framework Design Guidelines [Електронний ресурс] – Режим доступу до ресурсу: https://msdn.microsoft.com/en-us/library/ms229042.aspx.

16. Get started with Azure Blob storage using .NET [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/azure/storage/storage-dotnet-how-to-use-blobs.

17. Girish L S. Building Private Cloud using OpenStack. / Girish L S., Dr. H. S. Guruprasad. // International Journal of Emerging Trends & Technology in Computer Science, 2014 – pp. 134 – 138

18. Introduction [Електронний ресурс] – Режим доступу до ресурсу: https://www.rabbitmq.com/tutorials/tutorial-one-dotnet.html.

19. Introduction to ASP.NET Core [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/aspnet/core/.

20. JSON Web Token (JWT) [Електронний ресурс] – Режим доступу до ресурсу: https://tools.ietf.org/html/rfc7519.

21. Kruchten P. The Rational Unified Process: An Introduction / Philippe Kruchten., 336.

22. MacDonald M. Windows Presentation Foundation in .NET 4.5 / Matthew MacDonald., 2012. – 1078 с.

23. Marc Farley. Rethinking Enterprise Storage: A Hybrid Cloud Model. / Marc Farley. // Microsoft Press, 2013 – pp. 24 – 28

24. Microservice Architecture / I.Nadareishvili, M. Ronnie, M. Matt, A. Mike., 2016. – 146 с. – (978-1-4919-5625-0).

25. Microsoft Office 365 [Електронний ресурс] – Режим доступу до ресурсу: https://products.office.com/en-us/office-365-personal.

26. RabbitMQ: Введение в AMQP [Електронний ресурс] – Режим доступу до ресурсу: https://habrahabr.ru/post/64192/.

27. Seshardi S., AngularJS: Up and Running. // Seshadri S., Green B. - O'Reilly Media, 2014 – 322 с.

28. The Incremental Commitment Spiral Model: Principles and Practices for Successful Systems and Software / B.Boehm, J. Lane, S. Koolmanojwong, R. Turner., 2014. – 336 с.

29. The Repository Pattern [Електронний ресурс] – Режим доступу до ресурсу: https://msdn.microsoft.com/en-us/library/ff649690.aspx.

30. Top 10 payment gateways you should choose for your E-Commerce store [Електронний ресурс] – Режим доступу до ресурсу: https://www.brainsins.com/en/blog/top-10-payment-gateways-e-commerce-us/3661.

31. Making the Jump from Coding Tutorials to MVC [Електронний ресурс] – Режим доступу до ресурсу: https://epabon.silvrback.com/making-the-jump-from-coding-tutorials-to-mvc.

32. Version Control Systems Popularity [Електронний ресурс] – Режим доступу до ресурсу: https://rhodecode.com/insights/version-control-systems-2016.

33. Visual Studio [Електронний ресурс] – Режим доступу до ресурсу: https://www.visualstudio.com/ru/vs/.

34. What is Azure? [Електронний ресурс] – Режим доступу до ресурсу: https://azure.microsoft.com/en-ca/overview/what-is-azure/.

35. What is horizontal scalability (scaling out) [Електронний ресурс] – Режим доступу до ресурсу: http://searchcio.techtarget.com/definition/horizontal-scalability

36. What is RAD model - advantages, disadvantages and when to use it? [Електронний ресурс] – Режим доступу до ресурсу: http://istqbexamcertification.com/what-is-rad-model-advantages-disadvantages-and-when-to-use-it/.

37. Williamson K., Learning AngularJS. // Williamson K. - O'Reilly Media, 2015 – 212 с.

Додаток А. Концепція сервісу програмного забезпечення

CONFERENCE PROCEEDINGS

МАТЕРІАЛИ КОНФЕРЕНЦІЇ

summer infoCom 2017

Summer InfoCom Advanced Solutions 2017

IV МІЖНАРОДНА НАУКОВО-ПРАКТИЧНА КОНФЕРЕНЦІЯ
1-2 червня 2017 року

ISBN 978-966-2344-54-7

Україна, Київ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНСТИТУТ МОДЕРНІЗАЦІЇ ЗМІСТУ ОСВІТИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

SUMMER INFOCOM ADVANCED SOLUTIONS 2017

МАТЕРІАЛИ

IV МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ

CONFERENCE PROCEEDINGS

IV SCIENTIFIC AND PRACTICAL CONFERENCE

КИЇВ, УКРАЇНА
1-2 ЧЕРВНЯ 2017 РОКУ

Концепція сервісу програмного забезпечення

Хмелюк Володимир Сергійович
ст. викладач
КПІ ім. Ігоря Сікорського
Україна, Київ

Майер Ілля Сергійович
студент
КПІ ім. Ігоря Сікорського
Україна, Київ

Озеракін Микита Дмитрович
студент
КПІ ім. Ігоря Сікорського
Україна, Київ

Тимошенко Олександр Олександрович
студент
КПІ ім. Ігоря Сікорського
Україна, Київ

Тези доповіді містять опис концепції програмного сервісу, як безперервного та безетапного еволюційного процесу надання програмних послуг, що заміняє стандартний життєвий цикл програмного забезпечення (планування, розробку, тестування, модифікацію, супровід) та підтримуючі процеси (оплату, інтеграцію, оренду, взаємодію клієнта та провайдера).

Ключові слова: програмне забезпечення, розробка, сервіс, провайдер, замовник

Завдяки планомірному розвитку та досягненням технічного прогресу багато промислових галузей переходять від продуктових політик до сервісних. Так, наприклад, замість того, щоб купити супутниковий телефон у певної компанії, вкласти з нею договір та оплачувати щомісячну абонплату (продуктовий підхід) – ви маєте змогу купити телефон будь-якого виробника на ваш розсуд, обрати компанію, що буде вам надавати послугу зв'язку (сервіс) і надалі купувати додаткові послуги у цієї компанії за необхідності, орієнтуючись лише на власні потреби та товщину гаманця. Або ж замість купувати собі електрогенератор, ви можете встановити собі лічильник використання електроенергії і платити за сервіс надання доступу до електричних мереж міста згідно обсягів використаної електроенергії та тарифу. Ваш автомобіль теж ймовірно обслуговується в сервісному відділі певного автосалону. Подібних прикладів існує безліч.

І, хоча сфера інформаційних технологій розвивається найбільш динамічно та зазвичай є форпостом запровадження різноманітних нововведень та новаторських рішень, з переходом від продуктового підходу до сервісного не все так гладко. Так в частині «хард» давно вже практикується надання сервісів оренди серверних потужностей, сервісів хмарних містилищ та хмарних обчислень, сервісів послуг інтернет-провайдерів, тощо. А от в частині «софт», тобто програмного забезпечення, – повна тиша. Ви маєте змогу лише купити програмний продукт, або замовити його розробку, що по суті одне й те ж. Правда деякі великі компанії розробники ПЗ, наприклад Microsoft, все ж намагаються запровадити ідею надання послуг використання ПЗ (той же Office 365), але такі виключення поодинокі і лише підтверджують сумне правило.

І, оскільки ще не існує типових сервісів програмного забезпечення[1], спробуємо з'ясувати, якими мають бути такі сервіси і яким вимогам вони мають задовольняти. Отож, основним видом сервісу програмного забезпечення (СПЗ) має стати сервіс безперервної розробки ПЗ (що має замінити собою весь життєвий цикл ПЗ: планування, розробку, тестування, модифікацію, супровід, оплату, тощо). Бо, хоча деякі іменити компанії-розробники наголошують, що вони надають сервіс ПЗ, насправді вони надають лише можливість вибору однієї з продуктових політик використання вже існуючого продукту.

Спробуємо уявити, як би могла виглядати типова взаємодія замовника послуги ПЗ (клієнта сервісу) та розробника ПЗ (в даному контексті провайдера сервісу):

А) Я клієнт:

Я реєструюся на сайті сервісу ПЗ як Замовник. Я описую функціонал, який я хочу отримувати від цільового ПЗ, орієнтовні терміни виходу на робочий режим ПЗ, допустиму вартість умовної години розробки та інші необов'язкові характеристики сервісу. Мою заявку зі статусом «Пошук виконавця» бачать всі зареєстровані розробники ПЗ (провайдери). Зацікавлені провайдери пропонують свої послуги (сервіс ПЗ). Обговорення деталей ведеться тут-же на сайті. В мене є можливість побачити контактні дані провайдерів та зв'язатися з ними безпосередньо для обговорення деталей та нюансів взаємодії. Після вивчення пропозицій я обираю провайдера та замовляю його послугу ПЗ, заключаючи з ним Договір та закриваючи цим свою заявку. З цього моменту мені надається сервіс ПЗ. Я оплачую обумовлену грошову суму на рахунок провайдера, якщо це передбачено (prepaid2). Далі в CS3 я створюю початкові вимоги до ПЗ. Провайдер оцінює мої вимоги в умовних трудовитратах. Я маю змогу деталізувати вимоги, змінити, відмінити, перевпорядкувати, тощо та запропонувати свою вартість реалізації кожної з вимог. Після змін у вимогах і я і провайдер можемо змінити власну запропоновану ціну реалізації вимог. Після певного ітеративного процесу узгодження вимог та їх вартості (так званий аукціон вимог) я підтверджую готовність обраних вимог до реалізації. Провайдер підтверджує факт запуску вимог у розробку.

[1] Під сервісом програмного забезпечення СПЗ надалі будемо розуміти повний комплекс робіт з підтримки життєвого циклу ПЗ в поєднанні з фінансовими аспектами розподілений в часі.

[2] Слово «prepaid» можна перекласти як «передплата». Якщо коротко, prepaid — це спосіб розрахунку з провайдером послуги. Традиційно існують два варіанти оплати надання послуг: кредитний і дебетний. Якщо користуєшся послугою, а потім здійснюєш оплату рахунків що надійшли — це кредитна система. Дебетна система початково передбачає наявність на вашому рахунку певної суми. Наприклад, авансову форму розрахунку можна віднести до дебетових систем: Ви кладете на рахунок певну суму і після цього її витрачаєте

[3] CS – абревіатура референсної програмної системи підтримки СПЗ CleanSlate, що розробляється за участі студентів та викладачів НТУУ «КПІ»

Після реалізації вимоги до ПЗ провайдер повідомляє мене про реалізацію вимоги в новій версії ПЗ, яка автоматично розгортається у мене підсистемою розгортання (згідно налаштувань розгортання та політик підтримки версійності ПЗ). Я підтверджую виконання вимоги в повному обсязі після перевірки функціонування ПЗ. В цей момент з мого рахунку списується оговорена грошова сума а вимога вважається закритою. В процесі моєї взаємодії з провайдером можуть виникати нові вимоги, вимоги можуть ієрархічно розбиватися на дрібніші, об'єднуватися, відмінятися і т.д. згідно алгоритму життєвого циклу вимог. Одними з підвидів вимог будуть вимоги на виправлення помилок, покращення або зміну функціоналу, документування, створення навчальних матеріалів та інші. Надання сервісу провайдером припиняється за двосторонньою згодою, або в випадках передбачених укладеним Договором. Типовим варіантом є нескінченне надання послуги. Переваги, які я отримую від використання концепції програмного сервісу:

- можливість доступу до сервісу ПЗ за допомогою різних робочих обчислювальних машин (ноутбук, смартфон);
- механізм пошуку та конкурсного вибору провайдера сервісу ПЗ;
- середовище для спілкування з провайдером щодо замовленого сервісу ПЗ;
- мінімізацію часу та зусиль на створення технічного завдання на розробку ПЗ, методики перевірок та тестувань та іншої супутньої документації;
- можливість формування поточних версій деяких паперових документів в будь-який час (наприклад ТЗ, звіти, тощо);
- прозорість виконаних робіт провайдером згідно вимогам;
- можливість надгнучкої зміни вимог до ПЗ;
- систему встановлення пріоритетів реалізації вимог узгоджену з провайдером;
- можливість та середовище обліку помилок та відстеження їх виправлення;
- володіння та доступ до містилища поточних напрацювань (вихідні коди, зкомпільовані модулі, документація, тощо);
- можливість використання професійного ревізора оцінок реалізації вимог до ПЗ;
- можливість повного або часткового залучення зацікавлених осіб в процес розробки зі сторони замовника згідно обраної рольової політики в межах надання сервісу ПЗ;
- спрощення розгортання та оновлення ПЗ;
- своєчасність та оперативність оплати робіт виконаних провайдером;
- відстеження та планування бюджету на сервіс ПЗ;
- можливість зміни провайдера в будь-який час без втрати існуючих напрацювань та історії змін вимог до ПЗ;

Б) Я провайдер:

Я реєструюся на сайті сервісу ПЗ як Провайдер. Я вказую профільні предметні області (в яких я маю певні напрацювання та/або конкурентні переваги), посилання на інформацію по реалізованим мною проектам, кваліфікацію та склад команди розробки та супроводження ПЗ, орієнтовну вартість години надання СПЗ, тощо. Мій профіль провайдера разом з рейтингом, відгуками та іншими оцінками бачать як всі зареєстровані клієнти СПЗ так і інші розробники ПЗ (провайдери). Зацікавлені клієнти звертаються до мене з метою уточнення деталей надання СПЗ для можливої подальшої співпраці. Інші провайдери мають змогу звернутися до мене як до співвиконавця з метою доручити мені виконання певних робіт в межах їх власного СПЗ (тобто реалізується платформа пошуку виконавців робіт, але з урахуванням їх планової зайнятості та інших факторів). В мене є можливість побачити контактні дані клієнтів (якщо вони не вказали зворотного) та зв'язатися з ними безпосередньо для обговорення пропозиції по цільовому СПЗ. Після обговорення пропозицій я вкладаю договір на надання СПЗ з клієнтами або іншими провайдерами. Я розпочинаю уточнення початкових вимог замовника до ПЗ за допомогою механізму «аукціон задач» на базовому рівні (тобто до рівня достатнього для планування початкових робіт). Після підтвердження клієнтом починаю виконання робіт по реалізації вказаних вимог. Після виконання вимоги очікую оцінки клієнта на відповідність реалізації. В разі підтвердження клієнтом повного виконання вимоги проводиться фіксація в системі інформації про елемент оплати (фізичне перерахування коштів з таким обліком напряму не зв'язане). Подальші вимоги до ПЗ з'являються в системі в процесі надання СПЗ. Вони можуть генеруватися замовником або розробником з обов'язковим затвердженням зацікавленими сторонами. Переваги, які я отримую від використання концепції програмного сервісу:

- можливість доступу до сервісу ПЗ за допомогою різних робочих обчислювальних машин (ноутбук, смартфон);
- механізм пошуку клієнтів або генеральних підрядників для сервісу ПЗ;
- середовище для спілкування з клієнтами щодо сервісу ПЗ, що надається;
- мінімізацію часу та зусиль на створення технічного завдання на розробку ПЗ, методики перевірок та тестувань та іншої супутньої документації;
- можливість формування поточних версій деяких паперових документів в будь-який час (наприклад звітність по виконанню робіт, часові графіки, іншу інформацію необхідну для подальшого планування та виконання робіт);
- можливість надгнучкої зміни вимог до ПЗ;
- систему встановлення пріоритетів реалізації вимог узгоджену з клієнтом;
- можливість та середовище обліку помилок та відстеження їх виправлення;
- можливість легкого залучення до виконання робіт інших провайдерів СПЗ в якості підрядників;
- спрощення розгортання та оновлення ПЗ;
- можливість передачі клієнта іншому провайдеру (за згоди клієнта) без втрати існуючих напрацювань та історії змін вимог до ПЗ;

- відстеження та часткове прогнозування фінансових надходжень;
- гарантування оплати виконаних робіт клієнтом незалежно від факту повної реалізації вимог у випадку зміни вимог клієнтом, або при припиненні робіт з ініціативи клієнта;
- гнучкі механізми прогнозування та відстеження часових термінів реалізації вимог, задач, робіт, тощо;
- мінімізація бюрократичних витрат;
- значне підвищення взаєморозуміння з клієнтом та максимальне уникнення конфліктних ситуацій;
- реалізація процедур та механізмів можливого вирішення конфліктів за допомогою залучення третьої сторони;
- середовище тісної інтеграції зовнішніх систем розробки та супроводження ПЗ клієнтів в спеціалізовані АРМ провайдера;

Впровадження концепції сервісу програмного забезпечення в життя передбачає проведення низки організаційних заходів, розробки нових та уточнення існуючих методологій розробки та супроводження ПЗ, а також створення програмного забезпечення для підтримки середовища функціонування бізнес-площадок надання СПЗ.

Першою референсною системою підтримки СПЗ є програмна система CleanSlate, що розробляється за участі студентів та викладачів НТУУ «КПІ» та буде детально розглянута в низці наступних тематичних публікацій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Microsoft Office 365 [Електронний ресурс] — Режим доступу до ресурсу: https://products.office.com/en-us/office-365-personal.

2. Лармен К. Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum / Крейг Лармен., 2008. — 368 с. — (1-ше). — (978-0321480965).

3. Андерсон Д. Kanban: Successful Evolutionary Change for Your Technology Business / Девід Андерсон., 2010. — 278 с. — (978-0984521401)

4. Кніберг Г. Scrum and XP from the Trenches / Генрік Кніберг., 2007. — 142 с. — (2-ге). — (978-1-4303-2264-1).

ISSN 2520-2057

МІЖНАРОДНИЙ НАУКОВИЙ ЖУРНАЛ «ІНТЕРНАУКА»

INTERNATIONAL SCIENTIFIC JOURNAL «INTERNAUKA»

МЕЖДУНАРОДНЫЙ НАУЧНЫЙ ЖУРНАЛ «ИНТЕРНАУКА»

№ 8 (48) / 2018
1 том

МІЖНАРОДНИЙ НАУКОВИЙ ЖУРНАЛ «ІНТЕРНАУКА»

INTERNATIONAL SCIENTIFIC JOURNAL «INTERNAUKA»

МЕЖДУНАРОДНЫЙ НАУЧНЫЙ ЖУРНАЛ «ИНТЕРНАУКА»

Свидетельство
о государственной регистрации
печатного средства массовой информации
КВ № 22444-12344ПР

Сборник научных трудов

№ 8 (48)

1 том

Киев 2018

№ 8 (48) 1 т. 2018 квітень
МІЖНАРОДНИЙ НАУКОВИЙ ЖУРНАЛ «ІНТЕРНАУКА»
INTERNATIONAL SCIENTIFIC JOURNAL «INTERNAUKA»
МЕЖДУНАРОДНЫЙ НАУЧНЫЙ ЖУРНАЛ «ИНТЕРНАУКА»

ЗМІСТ
CONTENTS
СОДЕРЖАНИЕ

УДК 004.891.3:004.3

Майер Ілля Сергійович
студент кафедри автоматизації та управління в технічних системах
факультету інформатики та обчислювальної техніки
Національного технічного університету України
«Київський політехнічний інститут імені Ігоря Сікорського»
Майер Илья Сергеевич
студент кафедры автоматизации и управления в технических системах
факультета информатики и вычислительной техники
Национального технического университета Украины
«Киевский политехнический институт имени Игоря Сикорского»
Maier Illia
Student of the Department of Automation and Control in Technical Systems of the
Faculty of Computer Science and Computer Engineering of the
National Technical University of Ukraine
«Igor Sikorsky Kyiv Polytechnic Institute»

Хмелюк Марина Сергіївна
асистент кафедри автоматизації та управління в технічних системах
факультету інформатики та обчислювальної техніки
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Хмелюк Марина Сергеевна
ассистент кафедры автоматизации и управления в технических системах
факультета информатики и вычислительной техники
Национальный технический университет Украины
«Киевский политехнический институт имени Игоря Сикорского»
Khmeljuk Marina
Assistant of the Department of Automation and Control in Technical Systems
of the Faculty of Computer Science and Computer Engineering
National Technical University of Ukraine
«Igor Sikorsky Kyiv Polytechnic Institute»

ПІДТРИМКА ЖИТТЄВОГО ЦИКЛУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПОДДЕРЖКА ЖИЗНЕННОГО ЦИКЛА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ
SOFTWARE LIFE CYCLE SUPPORT

Анотація. *Процес сучасної розробки програмного забезпечення орієнтований на життєвий цикл програмного продукту. Всі існуючі в даний час технології, методики та стандарти безпосередньо або опосередковано стосуються або регламентують етапи життєвого циклу, як за функціональним наповненням, так і за змістом. Процес розробки програмних систем тісно пов'язаний з областю управління проектами, тому що будь-який програмний продукт є унікальним результатом. Від організації цього процесу безпосередньо залежать основні характеристики виконання програмного проекту – терміни виконання, запланований бюджет, якість готового продукту. Але професійне управління проектами саме по собі не може забезпечити досягнення зазначених характеристик. Важливу роль у цьому відіграє архітектура програмної системи, досвід і кваліфікація учасників команди розробки, а також правильне документування всіх процесів розробки програмного забезпечення.*

Ключові слова: *програмний сервіс, життєвий цикл програмного забезпечення.*

Аннотация. *Процесс современной разработки программного обеспечения ориентирован на жизненный цикл программного продукта. Все существующие в настоящее время технологии, методики и стандарты непосредственно или косвенно касаются или регламентирующих этапы жизненного цикла, как по функциональному наполнению, так и по содержанию. Процесс разработки программных систем тесно связан с областью управления проектами, потому что любой программный продукт является уникальным результатом. От организации этого процесса напрямую зависят основные характеристики выполнения программного проекта – сроки выполнения, запланированный бюджет, качество готового продукта. Но профессиональное управление проектами само по себе не может обеспечить достижения указанных характеристик. Важную роль в этом играет архитектура программной системы, опыт и квалификация участников команды разработки, а также правильное документирование всех процессов разработки программного обеспечения.*

Ключевые слова: *программный сервис, жизненный цикл программного обеспечения.*

Summary. *The process of modern software development focuses on the life cycle of a software product. All currently existing technologies, techniques and standards directly or indirectly relate or regulate the stages of the life cycle, both in terms of functional content and content. The process of developing software systems is closely linked to the project management area, because any software product is a unique result. From the organization of this process directly depend on the main characteristics of the implementation of the program project – the timing, the planned budget, the quality of the finished product. But professional project management alone can not achieve the achievement of these characteristics. An important role in this is played by the architecture of the software system, the experience and qualifications of the team development team, as well as the proper documentation of all software development processes.*

Key words: *software service, software life cycle.*

Постановка проблеми. Комп'ютеризація в світі збільшує свою швидкість з кожним днем, а кількість розроблюваного програмного забезпечення збільшується з ще більшою швидкість, що створює велике навантаження на керуючі гілки відповідних підрозділів.

Аналіз останніх досліджень і публікацій. Дослідження базується на працях таких видатних авторів у галузі розробки програмного забезпечення як К. Петерсона [1], Д. Шора [2], Л. Марка [3], К. Бека [4], Б. Бохема [5].

Формулювання цілей статті (постановка завдання). Аналіз різнобічних наукових праць та досліджень. Виділення найсильніших рис різних підходів до розробки програмного забезпечення.

Виклад основного матеріалу. В реальних умовах проектування інформаційних систем — це пошук способу, який забезпечить необхідну функціональність системи засобами існуючих технологій з урахуванням встановлених обмежень.

Під методологією розробки розуміють набір методів та критеріїв оцінки, які використовуються для постановки задачі, планування, контролю та в кінцевому підсумку — для досягнення поставленої цілі. Сам процес розробки описується моделлю, котра визначає послідовність найбільш загальних етапів та очікуваних результатів.

Зараз існує досить багато методологій управління проектами та відповідного програмного забезпечення. Всі вони мають свої переваги та недоліки. Розглянемо деякі основні методологій, які довели свою ефективність при розробці ПЗ.

1. Каскадна модель («Waterfall Model»)

Каскадна модель одна з найбільш традиційних та загально використовуємих методологій для програмної розробки. Ця модель життєвого циклу,, зазвичай вважається як класичний стиль програмної розробки. Вона висвітлює процес програмної розробки як лінійну послідовну течію — під цим розуміють, що будь-яка фаза в процесі розробки починається тільки якщо попередня фаза завершена. Цей підхід не надає можливості повертатися назад до попередньої фази, щоб внести зміни в вимогах. Цю методолгію застосовують коли вимоги вже обговорені, немає проблем з кваліфікованими фахівцями, у відносно невеликих проектах.

Переваги:

- каскадна модель дуже просто та легка для розуміння та використання, що дійсно добре для новачків-розробників;
- в цій моделі фази виконуються та завершуються лише один раз — це суттєво зберігає велику кількість часу;
- цей підхід до розробки більш ефективен на невеликих проектах, де вимоги дуже добре сформульовані;

Недоліки:

- ця модель може використовуватися тільки коли чітко визначені попередні вимоги;
- ця модель не прийнятна для підтримки проектів;
- головний недолік цього методу якщо додаток знаходиться в стадії тестування немає можливості повернутуся до попереднього етапу, щоб внести деякі зміни;
- немає можливості показувати працюючий додаток доки не досягнуто останньої стадії циклу;
- не ідеальна для проектів, де вимоги недостатньо визначені та мають багато місць для зміни.

2. Гнучка методологія розробки («Agile model»)

Гнучка методологія використовується для проектування впорядкованого управління процесом

розробки котрий дозволяє вносити постійні зміни в розробку проекту. Ця модель використовується для максимального зменшення ризику при розробці продукту в короткі часові проміжки котрі називаються ітераціями і зазвичай тривають від одного тижня до одного місяця.

Цю модель слід застосовувати коли потреби користувачів постійно змінюються в динамічному бізнесі. Зміни на Agile реалізуються за меншу ціну із-за постійних спринтів. На відміну від каскадної моделі, в гнучкій моделі для старту проекту досить лише невеликого планування.

Переваги:

– гнучка методологія має адаптивний підхід котрий дозволяє змінювати вимоги клієнтів;
– безпосередній зв'язок та постійні відгуки замовників або їх представників не залишає місця для невизначенностей.

Недоліки:

– ця методологія зосереджена на створенні програмного забезпечення раніше, ніж документації, звідси може бути нестача документації;
– процес розробки може вийти з під контролю, якщо замовник чітко не представляє кінцевий результат проекту.

3. Спіральна модель («Spiral model»)

Спіральна модель є досить комплексною моделлю що зосереджується на ранньому виявленні та зменшенні ризиків проектів. В цій методології розробники починають з малих масштабів потім виявляють можливі ризики в проекті, складаються план для запобігання їх і в завершенні вирішують чи варто переходити до наступної стадії проекту, щоб зробити наступну ітерацію спіралі. Успіх будь-якого життєвого циклу спіральної моделі залежить від надійного, уважного та грамотного керівництва проекту.

Ця модель не підійде для малих проектів, вона резонна для складних і дорогих, наприклад, таких, як розробка системи документообліку для банку, коли кожен наступний крок вимагає більшого аналізу для оцінки наслідків, ніж програмування.

Переваги:

– об'ємний аналіз ризиків звичайно ж зменшує можливість провалу
– ця модель досить добре підходить для великих та ризикованих проектів;
– в спіральній моделі, додаткову функціональність можна додати пізніше;
– більш задовільна для високоризикованих проектів, де бажання та потреби замовника можуть змінюватися час від часу.

Недоліки:

– досить витратна модель з точки зору розробки;
– в цілому успіх проекту залежить від фази аналізу ризиків, невдача в цій фазі може спричинити провал проекту;
– не підходить для низькоризикованих проектів;

4. Екстремальне програмування («Extreme Programming»)

Екстремальне програмування — гнучка методологія розробки програмного забезпечення. Ця методологія, відома ще як «XP», в основному використовується для створення додатків в дуже нестабільному середовищі. Це додає великою гнучкості під час процесу моделювання. Головна перевага цієї методології — це те, що вона досить малозатратна. В моделі XP доволі часто буває, що вартість змін вимог на більш пізній стадії проекту може бути досить високою.

Переваги:

– методологія екстремального програмування робить акцент на залученості замовника;
– ця модель допомагає визначити доцільні плани і графіки та дає можливість розробникам самостійно зафіксувати їхні графіки що є безумовно найбільшою перевагою;

Недоліки:

– ця методологія ефективна лише якщо розробники повністю зосереджені та захоплені розробкою;
– ця модель вимагає занадто багато змін в розробці, що дуже трудозатратно для розробника;
– в цій методології, як правило, неможливо визначити точні трудозатрати тому, що на початку проекту ніхто не уявляє цілий об'єм робіт та вимог.

Порівняння методологій

Найбільш популярними та одними із батьків усіх інших є каскадна та Agile методології. З однієї сторони (каскадна) ми плануємо все до найменших деталей, визначаємо жорсткі терміни, маємо фіксований бюджет, але проблеми приходять коли потрібно зробити якісь правки або взагалі змінити русло куди йде розробка програмного продукту. З іншої сторони (Agile) маємо гнучку систему спринтів, замовник платить за спринти, досить легко вносити зміни, але досить складно встановлювати терміни для випуску продукту та важко прорахувати бюджет. У таблиці 1 проведено порівняння вищезгаданих методологій за певними критеріями.

Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі. Зростаючи кожного дня попит на програмне забезпечення створює необхідність правильної постановки процесів в підрозділах розробки, аби вони могли задовольняти потреби бізнесу та створювати якісний, підтримуваний та функціональний продукт. Існуючі методології розробки були створені та перевірені часом, але їх недоліки інколи є критичними для їх користувачів, що відкриває простір для покращення та створення новий підходів до розробки програмного забезпечення.

Таблиця 1

Порівняння методологій

	Каскадна	Agile	Спіральна	XP
Детермінованість вимог	повністю	основні	бажано основні	початкові
Розмір проекту	нижче середн.	великий	великий	нижче середн.
Контроль витрат	високий	низький	нижче середн.	низька
Гарантія успіху	невисока	висока	висока	середня
Рівень ризику	високий	низький	низький	вище середн.
Залученість замовника	низька	вище середн.	середня	висока
Простота використання	висока	вище середн.	нижче середн.	низька
Повернення до попередньої фази	–	+	–/+	+

Література

1. Петерсон К. The Waterfall Model in Large-Scale Development / Кай Петерсон. — 386 с.
2. Шор Д. The Art of Agile Development / Джеймс Шор., 2007. — 440 с.
3. Марк Л. Agile Project Management For Dummies / Лейтон Марк. — Хобокен, NJ: For Dummies, 2012. — 360 с.
4. Бек К. Extreme Programming Explained: Embrace Change / К. Бек, К. Андрес. — 224 с.
5. Бохем Б. The Incremental Commitment Spiral Model: Principles and Practices for Successful Systems and Software / Б. Бохем, Д. Лейн, С. Кулманойвонг, Р. Тьорнер., 2014. — 336 с.

References

1. Peterson K. The Waterfall Model in Large-Scale Development / Cay Peterson. — 386 p.
2. Shore D. The Art of Agile Development / James Shore, 2007. — 440 p.
3. Mark L. Agile Project Management For Dummies / Leyton Mark. — Hoboken, NJ: For Dummies, 2012. — 360 p.
4. Beck K. Extreme Programming Explained: Embrace Change / K. Beck, C. Andres. — 224 p.
5. Boehm B. The Incremental Commitment Spiral Model: Principles and Practices for Successful Systems and Software / B. Boehm, J. Lane, S. Koolmanojwong, R. Turner, 2014. — 336 p.

МІЖНАРОДНИЙ НАУКОВИЙ ЖУРНАЛ "ІНТЕРНАУКА"
INTERNATIONAL SCIENTIFIC JOURNAL"INTERNAUKA"

(Свідоцтво про реєстрацію: КВ № 22444-12344ПР)

СВІДОЦТВО про публікацію

Серія: МН *№ 76322*

засвідчує, що стаття

Майера Іллі Сергійовича

на тему:

«ПІДТРИМКА ЖИТТЄВОГО ЦИКЛУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»

опублікована в
Міжнародному науковому журналі "Інтернаука"
(INTERNATIONAL SCIENTIFIC JOURNAL "INTERNAUKA")

Web-адреса публікації:
https://www.inter-nauka.com/issues/2018/8/3689/

Головний редактор *Д. І. Коваленко*

Додаток В. Вихідний код модульного тесту

```
[TextFixture]
public class RequirementControllerTests : ControllerBaseTests
{
	private Mock<IRequirementManager> _mockRequirementMng;
	private Mock<RequirementController> _mockRequirementController;

	[SetUp]
	public void Setup
	{
		_mockRequirementMng = new Mock<IRequirementManager>(MockUnitOfWork);
		_mockRequirementController = new Mock<RequirementController>(_mockRequirementMng);
	}

	[Test]
	public void Requirement_Should_Create_One()
	{
		//Arrange
		var requirements = new List<Requirement>();
		requirements.Add(new Requirement());
		_mockRequirementController.Setup(x => x.Post).Returns(new JsonResult(requirements))

		//Act
		var toCrt = new CreateRequirementRequest();
		_mockRequirementMng.Create(toCrt);
```

```
            //Assert
            var reqs = MockUnitOfWork.RequirementRepository.GetAll();
            var res = (_mockRequirementController.Get() as JsonResult).Data as
List<Requirement>;
            Assert.AreEqual(reqs.Count, res.Count);
        }

        [Test]
        public void Requirement_Should_Create_WithParrent()
        {
            //Arrange
            var parrentId = Guid.NewGuid();
            var reqId = Guid.NewGuid();
            var requirements = new List<Requirement>();
            var req = new Requirement();

            req.Id = reqId;
            req.ParrentRequirementId = parrentId;
            requirements.Add(req);
            _mockRequirementController.Setup(x => x.Get).Returns(new
JsonResult(_mockRequirementMng.GetById(reqId)))

            //Act
            var toCrt = new CreateRequirementRequest();
            toCrt.ParrentRequirementId = parrentId;
            _mockRequirementMng.Create(toCrt);

            //Assert
```

```
        Assert.IsNotNull(_mockRequirementController.Get(reqId).ParrentRequirement);
        }

        [Test]
        public void Requirement_Should_Return_All()
        {
                //Arrange
                var reqs = new List<Requirement>();
                reqs.Add(new Requirement());
                reqs.Add(new Requirement());
                reqs.Add(new Requirement());
                _mockRequirementController.Setup(x => x.Get).Returns(new
JsonResult(reqs));

                //Act
                MockUnitOfWork.RequirementRepository.AddRange(reqs);
                var exReqs = _mockRequirementMng.GetAll();
                var res = (_mockRequirementController.Get() as JsonResult).Data as
List<Requirement>;

                //Assert
                Asser.AreEqual(exReqs.Count, res.Count)
        }

        [Test]
        public void Requirement_Should_Return_One()
        {
                //Arrange
```

```
            var requirements = new List<Requirement>();
            requirements.Add(new Requirement());
            _mockRequirementController.Setup(x => x.Post).Returns(new
JsonResult(requirements))
                        //Act
            var toCrt = new CreateRequirementRequest();
            _mockRequirementMng.Create(toCrt);

            //Assert
            var reqs = MockUnitOfWork.RequirementRepository.GetAll();
            var res = (_mockRequirementController.Get() as JsonResult).Data as
List<Requirement>;
            Assert.AreEqual(reqs.Count, 1);
        }
}
```

Додаток Г. Вихідний код основних компонентів

```
public class UserStore : IUserStore
{
    private readonly IUnitOfWork _unitOfWork;

    public UserStore(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
    }

                public Task<string> GetUserIdAsync(User user, CancellationToken
cancellationToken)
    {
        CheckUser(user, cancellationToken);
        return ParallelTask.FromResult(user.Id.ToString());
    }

                public Task<string> GetUserNameAsync(User user, CancellationToken
cancellationToken)
    {
        CheckUser(user, cancellationToken);
        return ParallelTask.FromResult(user.UserName);
    }

                public ParallelTask SetUserNameAsync(User user, string userName,
CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);
```

```
        user.UserName = userName;
        return ParallelTask.CompletedTask;
    }

     public Task<string> GetNormalizedUserNameAsync(User user, CancellationToken
cancellationToken)
    {
        CheckUser(user, cancellationToken);
        return ParallelTask.FromResult(user.NormalizedUserName);
    }

                public ParallelTask SetNormalizedUserNameAsync(User user, string
normalizedName, CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);

        user.NormalizedUserName = normalizedName;
        return ParallelTask.CompletedTask;
    }

       public async Task<IdentityResult> CreateAsync(User user, CancellationToken
cancellationToken)
    {
        CheckUser(user, cancellationToken);

        _unitOfWork.UserRepository.Add(user);
        await _unitOfWork.SaveChangesAsync(cancellationToken);

        return IdentityResult.Success;
    }
```

```
        public async Task<IdentityResult> DeleteAsync(User user, CancellationToken
cancellationToken)
    {
      CheckUser(user, cancellationToken);

      _unitOfWork.UserRepository.Delete(user);
      await _unitOfWork.SaveChangesAsync(cancellationToken);

      return IdentityResult.Success;
    }

            public Task<User> FindByIdAsync(string userId, CancellationToken
cancellationToken)
    {
      cancellationToken.ThrowIfCancellationRequested();
              return _unitOfWork.UserRepository.GetByIdAsync(Guid.Parse(userId),
cancellationToken);
    }

            public Task<User> FindByNameAsync(string normalizedUserName,
CancellationToken cancellationToken)
    {
      cancellationToken.ThrowIfCancellationRequested();
                                                                    return
_unitOfWork.UserRepository.FindByNormalizedUserNameAsync(cancellationToken,
normalizedUserName);
    }

        public async Task<IdentityResult> UpdateAsync(User user, CancellationToken
cancellationToken)
    {
```

```
        CheckUser(user, cancellationToken);

        _unitOfWork.UserRepository.Update(user);
        await _unitOfWork.SaveChangesAsync(cancellationToken);

        return IdentityResult.Success;
    }

    public Task<IList<User>> GetUsersForClaimAsync(Claim claim, CancellationToken
cancellationToken)
    {
                                                                    return
ParallelTask.FromResult(_unitOfWork.UserClaimRepository.FindUsersByClaim(claim));
    }

        public ParallelTask AddClaimsAsync(User user, IEnumerable<Claim> claims,
CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);

        claims.Select(x => new UserClaim { ClaimType = x.Type, ClaimValue = x.Value,
UserId = user.Id })
            .ForEach(_unitOfWork.UserClaimRepository.Add);

        return ParallelTask.CompletedTask;
    }

        public Task<IList<Claim>> GetClaimsAsync(User user, CancellationToken
cancellationToken)
    {
        CheckUser(user, cancellationToken);
```

```
                                                                    return
ParallelTask.FromResult<IList<Claim>>(_unitOfWork.UserClaimRepository.FindClaims
ByUserId(user.Id).Select(x => new Claim(x.ClaimType, x.ClaimValue)).ToList());
    }

      public ParallelTask RemoveClaimsAsync(User user, IEnumerable<Claim> claims,
CancellationToken cancellationToken)
    {
      CheckUser(user, cancellationToken);

      foreach (var claim in claims)
      {
                                                          var userClaims =
_unitOfWork.UserClaimRepository.FindClaimsByUserIdAndClaim(user.Id, claim);
        _unitOfWork.UserClaimRepository.DeleteRange(userClaims);
      }

      return ParallelTask.CompletedTask;
    }

      public ParallelTask ReplaceClaimAsync(User user, Claim claim, Claim newClaim,
CancellationToken cancellationToken)
    {
      CheckUser(user, cancellationToken);

                                                          var userClaims =
_unitOfWork.UserClaimRepository.FindClaimsByUserIdAndClaim(user.Id, claim);
      userClaims.ForEach(uc =>
                {
                  uc.ClaimType = newClaim.Type;
```

```
                    uc.ClaimValue = newClaim.Value;
                    _unitOfWork.UserClaimRepository.Update(uc);
                });

        return ParallelTask.CompletedTask;
    }

        public ParallelTask AddLoginAsync(User user, UserLoginInfo login,
CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);

        var externalLogin = new ExternalLogin
                {
                    LoginProvider = login.LoginProvider,
                    ProviderKey = login.ProviderKey,
                    ProviderDisplayName = login.ProviderDisplayName,
                    User = user
                };

        _unitOfWork.ExternalLoginRepository.Add(externalLogin);

        return ParallelTask.CompletedTask;
    }

        public Task<User> FindByLoginAsync(string loginProvider, string providerKey,
CancellationToken cancellationToken)
    {
                                                                    return
ParallelTask.FromResult(_unitOfWork.ExternalLoginRepository.GetUserByProviderAnd
Key(loginProvider, providerKey));
```

```
    }

     public Task<IList<UserLoginInfo>> GetLoginsAsync(User user, CancellationToken
cancellationToken)
    {
       CheckUser(user, cancellationToken);
       var logins = _unitOfWork.ExternalLoginRepository.GetLoginsByUserId(user.Id);

         return ParallelTask.FromResult<IList<UserLoginInfo>>(logins.Select(x => new
UserLoginInfo(x.LoginProvider, x.ProviderKey, x.User.UserName)).ToList());
    }

        public ParallelTask RemoveLoginAsync(User user, string loginProvider, string
providerKey, CancellationToken cancellationToken)
    {
       CheckUser(user, cancellationToken);
                                                          var login =
_unitOfWork.ExternalLoginRepository.GetByUserIdAndProviderAndKey(user.Id,
loginProvider, providerKey);
        _unitOfWork.ExternalLoginRepository.Delete(login);
       return ParallelTask.CompletedTask;
    }

                public Task<IList<User>> GetUsersInRoleAsync(string roleName,
CancellationToken cancellationToken)
    {
       cancellationToken.ThrowIfCancellationRequested();
       var role = _unitOfWork.RoleRepository.FindByNormalizedName(roleName);
```

```
                                                                    return
ParallelTask.FromResult(_unitOfWork.UserRoleProjectRepository.FindUsersByRoleId(ro
le.Id));
    }

        public async ParallelTask AddToRoleAsync(User user, string roleName,
CancellationToken cancellationToken)
    {
      CheckUser(user, cancellationToken);
      var role = _unitOfWork.RoleRepository.FindByNormalizedName(roleName);
        var userRole = new UserRoleProject { Id = Guid.NewGuid(), RoleId = role.Id,
UserId = user.Id };

      _unitOfWork.UserRoleProjectRepository.Add(userRole);

      await _unitOfWork.SaveChangesAsync(cancellationToken);
    }

        public Task<IList<string>> GetRolesAsync(User user, CancellationToken
cancellationToken)
    {
      CheckUser(user, cancellationToken);

      var roles = _unitOfWork.UserRoleProjectRepository.FindRolesByUserId(user.Id);

                return ParallelTask.FromResult<IList<string>>(roles.Select(x =>
x.Name).ToList());
    }

    public Task<bool> IsInRoleAsync(User user, string roleName, CancellationToken
cancellationToken)
```

```
    {
        CheckUser(user, cancellationToken);
        var role = _unitOfWork.RoleRepository.FindByNormalizedName(roleName);

                                                                    return
ParallelTask.FromResult(_unitOfWork.UserRoleProjectRepository.HasUserRole(user.Id,
role.Id));
    }

            public ParallelTask RemoveFromRoleAsync(User user, string roleName,
CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);
        var role = _unitOfWork.RoleRepository.FindByNormalizedName(roleName);
                                                        var userRole =
_unitOfWork.UserRoleProjectRepository.FindByUserIdAndRoleId(user.Id, role.Id);

        _unitOfWork.UserRoleProjectRepository.Delete(userRole);
        return _unitOfWork.SaveChangesAsync(cancellationToken);
    }

            public Task<string> GetPasswordHashAsync(User user, CancellationToken
cancellationToken)
    {
        CheckUser(user, cancellationToken);
        return ParallelTask.FromResult(user.PasswordHash);
    }

                public Task<bool> HasPasswordAsync(User user, CancellationToken
cancellationToken)
    {
```

```
        CheckUser(user, cancellationToken);
        return ParallelTask.FromResult(!string.IsNullOrWhiteSpace(user.PasswordHash));
    }

        public ParallelTask SetPasswordHashAsync(User user, string passwordHash,
CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);
        user.PasswordHash = passwordHash;

        return ParallelTask.CompletedTask;
    }

        public Task<string> GetSecurityStampAsync(User user, CancellationToken
cancellationToken)
    {
        CheckUser(user, cancellationToken);

        return ParallelTask.FromResult(user.SecurityStamp);
    }

            public ParallelTask SetSecurityStampAsync(User user, string stamp,
CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);

        user.SecurityStamp = stamp;
        return ParallelTask.CompletedTask;
    }
```

```
        public ParallelTask SetEmailAsync(User user, string email, CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);
        user.Email = email;

        return ParallelTask.CompletedTask;
    }

                public Task<string> GetEmailAsync(User user, CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);
        return ParallelTask.FromResult(user.Email);
    }

        public Task<bool> GetEmailConfirmedAsync(User user, CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);
        return ParallelTask.FromResult(user.EmailConfirmed);
    }

        public ParallelTask SetEmailConfirmedAsync(User user, bool confirmed, CancellationToken cancellationToken)
    {
        CheckUser(user, cancellationToken);
        user.EmailConfirmed = confirmed;

        return ParallelTask.CompletedTask;
    }
```

```
 public Task<User> FindByEmailAsync(string normalizedEmail, CancellationToken cancellationToken)
    {
                                                                    return ParallelTask.FromResult(_unitOfWork.UserRepository.FindByNormalizedEmail(normalizedEmail));
    }

        public Task<string> GetNormalizedEmailAsync(User user, CancellationToken cancellationToken)
    {
       CheckUser(user, cancellationToken);
       return ParallelTask.FromResult(user.NormalizedEmail);
    }

      public ParallelTask SetNormalizedEmailAsync(User user, string normalizedEmail, CancellationToken cancellationToken)
    {
       CheckUser(user, cancellationToken);
       user.NormalizedEmail = normalizedEmail;

       return ParallelTask.CompletedTask;
    }

    public void Dispose()
    {

    }

    private void CheckUser(User user, CancellationToken cancellationToken)
```

```
    {
        cancellationToken.ThrowIfCancellationRequested();

        if (user == null)
        {
            throw new ArgumentNullException(nameof(user));
        }
    }
}
```